```
::::::::::::::
Elab.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Import
import Lean.Elab.Exception
import Lean.Elab.Command
import Lean.Elab.Term
import Lean.Elab.App
import Lean.Elab.Binders
import Lean.Elab.LetRec
import Lean.Elab.Frontend
import Lean.Elab.BuiltinNotation
import Lean.Elab.Declaration
import Lean.Elab.Tactic
import Lean.Elab.Match
-- HACK: must come after `Match` because builtin elaborators (for `match` in this case) do not take priorities
import Lean.Elab.Quotation
import Lean.Elab.Syntax
import Lean.Elab.Do
import Lean.Elab.StructInst
import Lean.Elab.Inductive
import Lean.Elab.Structure
import Lean.Elab.Print
import Lean.Elab.MutualDef
import Lean.Elab.PreDefinition
import Lean.Elab.Deriving
import Lean.Elab.DeclarationRange
import Lean.Elab.Extra
::::::::::::::
Elab/App.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.FindMVar
import Lean.Parser.Term
import Lean.Elab.Term
import Lean.Elab.Binders
import Lean.Elab.SyntheticMVars
```

```
namespace Lean.Elab.Term
open Meta

builtin_initialize elabWithoutExpectedTypeAttr : TagAttribute ←
  registerTagAttribute `elabWithoutExpectedType "mark that applications of the given declaration should be elaborated without the expect

def hasElabWithoutExpectedType (env : Environment) (declName : Name) : Bool :=
  elabWithoutExpectedTypeAttr.hasTag env declName

/--
  Auxiliary inductive datatype for combining unelaborated syntax
  and already elaborated expressions. It is used to elaborate applications. -/
inductive Arg where
  | stx  (val : Syntax)
  | expr (val : Expr)
  deriving Inhabited

instance : ToString Arg := ⟨fun
  | Arg.stx  val => toString val
  | Arg.expr val => toString val⟩

/-- Named arguments created using the notation `(x := val)` -/
structure NamedArg where
  ref  : Syntax := Syntax.missing
  name : Name
  val  : Arg
  deriving Inhabited

instance : ToString NamedArg where
  toString s := "(" ++ toString s.name ++ " := " ++ toString s.val ++ ")"

def throwInvalidNamedArg {α} (namedArg : NamedArg) (fn? : Option Name) : TermElabM α :=
  withRef namedArg.ref <| match fn? with
    | some fn => throwError "invalid argument name '{namedArg.name}' for function '{fn}'"
    | none    => throwError "invalid argument name '{namedArg.name}' for function"

/--
  Add a new named argument to `namedArgs`, and throw an error if it already contains a named argument
  with the same name. -/
def addNamedArg (namedArgs : Array NamedArg) (namedArg : NamedArg) : TermElabM (Array NamedArg) := do
  if namedArgs.any (namedArg.name == ·.name) then
    throwError "argument '{namedArg.name}' was already set"
  return namedArgs.push namedArg

private def ensureArgType (f : Expr) (arg : Expr) (expectedType : Expr) : TermElabM Expr := do
  let argType ← inferType arg
```

```
  ensureHasTypeAux expectedType argType arg f

/-
  Relevant definitions:
  ```
  class CoeFun (α : Sort u) (γ : α → outParam (Sort v))
  abbrev coeFun {α : Sort u} {γ : α → Sort v} (a : α) [CoeFun α γ] : γ a
  ```
-/
private def tryCoeFun? (α : Expr) (a : Expr) : TermElabM (Option Expr) := do
  let v ← mkFreshLevelMVar
  let type ← mkArrow α (mkSort v)
  let γ ← mkFreshExprMVar type
  let u ← getLevel α
  let coeFunInstType := mkAppN (Lean.mkConst ``CoeFun [u, v]) #[α, γ]
  let mvar ← mkFreshExprMVar coeFunInstType MetavarKind.synthetic
  let mvarId := mvar.mvarId!
  try
    if (← synthesizeCoeInstMVarCore mvarId) then
      expandCoe <| mkAppN (Lean.mkConst ``coeFun [u, v]) #[α, γ, a, mvar]
    else
      return none
  catch _ =>
    return none

def synthesizeAppInstMVars (instMVars : Array MVarId) : TermElabM Unit :=
  for mvarId in instMVars do
    unless (← synthesizeInstMVarCore mvarId) do
      registerSyntheticMVarWithCurrRef mvarId SyntheticMVarKind.typeClass

namespace ElabAppArgs

/- Auxiliary structure for elaborating the application `f args namedArgs`. -/
structure State where
  explicit          : Bool -- true if `@` modifier was used
  f                 : Expr
  fType             : Expr
  args              : List Arg            -- remaining regular arguments
  namedArgs         : List NamedArg       -- remaining named arguments to be processed
  ellipsis          : Bool := false
  expectedType?     : Option Expr
  etaArgs           : Array Expr   := #[]
  toSetErrorCtx     : Array MVarId := #[] -- metavariables that we need the set the error context using the application being built
  instMVars         : Array MVarId := #[] -- metavariables for the instance implicit arguments that have already been processed
  -- The following field is used to implement the `propagateExpectedType` heuristic.
  propagateExpected : Bool  -- true when expectedType has not been propagated yet

```

```
abbrev M := StateRefT State TermElabM

/- Add the given metavariable to the collection of metavariables associated with instance-implicit arguments. -/
private def addInstMVar (mvarId : MVarId) : M Unit :=
  modify fun s => { s with instMVars := s.instMVars.push mvarId }

/-
  Try to synthesize metavariables are `instMVars` using type class resolution.
  The ones that cannot be synthesized yet are registered.
  Remark: we use this method before trying to apply coercions to function. -/
def synthesizeAppInstMVars : M Unit := do
  let s ← get
  let instMVars := s.instMVars
  modify fun s => { s with instMVars := #[] }
  Lean.Elab.Term.synthesizeAppInstMVars instMVars

/- fType may become a forallE after we synthesize pending metavariables. -/
private def synthesizePendingAndNormalizeFunType : M Unit := do
  synthesizeAppInstMVars
  synthesizeSyntheticMVars
  let s ← get
  let fType ← whnfForall s.fType
  if fType.isForall then
    modify fun s => { s with fType := fType }
  else
    match (← tryCoeFun? fType s.f) with
    | some f =>
      let fType ← inferType f
      modify fun s => { s with f := f, fType := fType }
    | none =>
      for namedArg in s.namedArgs do
        let f := s.f.getAppFn
        if f.isConst then
          throwInvalidNamedArg namedArg f.constName!
        else
          throwInvalidNamedArg namedArg none
      throwError "function expected at{indentExpr s.f}\nterm has type{indentExpr fType}"

/- Normalize and return the function type. -/
private def normalizeFunType : M Expr := do
  let s ← get
  let fType ← whnfForall s.fType
  modify fun s => { s with fType := fType }
  pure fType

/- Return the binder name at `fType`. This method assumes `fType` is a function type. -/
private def getBindingName : M Name := return (← get).fType.bindingName!
```

```
/- Return the next argument expected type. This method assumes `fType` is a function type. -/
private def getArgExpectedType : M Expr := return (← get).fType.bindingDomain!

def eraseNamedArgCore (namedArgs : List NamedArg) (binderName : Name) : List NamedArg :=
  namedArgs.filter (·.name != binderName)

/- Remove named argument with name `binderName` from `namedArgs`. -/
def eraseNamedArg (binderName : Name) : M Unit :=
  modify fun s => { s with namedArgs := eraseNamedArgCore s.namedArgs binderName }

/-
  Add a new argument to the result. That is, `f := f arg`, update `fType`.
  This method assumes `fType` is a function type. -/
private def addNewArg (arg : Expr) : M Unit :=
  modify fun s => { s with f := mkApp s.f arg, fType := s.fType.bindingBody!.instantiate1 arg }

/-
  Elaborate the given `Arg` and add it to the result. See `addNewArg`.
  Recall that, `Arg` may be wrapping an already elaborated `Expr`. -/
private def elabAndAddNewArg (arg : Arg) : M Unit := do
  let s ← get
  let expectedType ← getArgExpectedType
  match arg with
  | Arg.expr val =>
    let arg ← ensureArgType s.f val expectedType
    addNewArg arg
  | Arg.stx val  =>
    let val ← elabTerm val expectedType
    let arg ← ensureArgType s.f val expectedType
    addNewArg arg

/- Return true if the given type contains `OptParam` or `AutoParams` -/
private def hasOptAutoParams (type : Expr) : M Bool := do
  forallTelescopeReducing type fun xs type =>
    xs.anyM fun x => do
      let xType ← inferType x
      return xType.getOptParamDefault?.isSome || xType.getAutoParamTactic?.isSome

/- Return true if `fType` contains `OptParam` or `AutoParams` -/
private def fTypeHasOptAutoParams : M Bool := do
  hasOptAutoParams (← get).fType

/- Auxiliary function for retrieving the resulting type of a function application.
   See `propagateExpectedType`.

   Remark: `(explicit : Bool) == true` when `@` modifier is used. -/
```

```lean
private partial def getForallBody (explicit : Bool) : Nat → List NamedArg → Expr → Option Expr
  | i, namedArgs, type@(Expr.forallE n d b c) =>
    match namedArgs.find? fun (namedArg : NamedArg) => namedArg.name == n with
    | some _ => getForallBody explicit i (eraseNamedArgCore namedArgs n) b
    | none =>
      if !explicit && !c.binderInfo.isExplicit then
        getForallBody explicit i namedArgs b
      else if i > 0 then
        getForallBody explicit (i-1) namedArgs b
      else if d.isAutoParam || d.isOptParam then
        getForallBody explicit i namedArgs b
      else
        some type
  | 0, [], type => some type
  | _, _,  _    => none

private def shouldPropagateExpectedTypeFor (nextArg : Arg) : Bool :=
  match nextArg with
  | Arg.expr _  => false -- it has already been elaborated
  | Arg.stx stx =>
    -- TODO: make this configurable?
    stx.getKind != ``Lean.Parser.Term.hole &&
    stx.getKind != ``Lean.Parser.Term.syntheticHole &&
    stx.getKind != ``Lean.Parser.Term.byTactic

/-
  Auxiliary method for propagating the expected type. We call it as soon as we find the first explicit
  argument. The goal is to propagate the expected type in applications of functions such as
  ```lean
  Add.add {α : Type u} : α → α → α
  List.cons {α : Type u} : α → List α → List α
  ```
  This is particularly useful when there applicable coercions. For example,
  assume we have a coercion from `Nat` to `Int`, and we have
  `(x : Nat)` and the expected type is `List Int`. Then, if we don't use this function,
  the elaborator will fail to elaborate
  ```
  List.cons x []
  ```
  First, the elaborator creates a new metavariable `?α` for the implicit argument `{α : Type u}`.
  Then, when it processes `x`, it assigns `?α := Nat`, and then obtain the
  resultant type `List Nat` which is **not** definitionally equal to `List Int`.
  We solve the problem by executing this method before we elaborate the first explicit argument (`x` in this example).
  This method infers that the resultant type is `List ?α` and unifies it with `List Int`.
  Then, when we elaborate `x`, the elaborate realizes the coercion from `Nat` to `Int` must be used, and the
  term
  ```
```
```

```
  @List.cons Int (coe x) (@List.nil Int)
  ```
  is produced.

  The method will do nothing if
  1- The resultant type depends on the remaining arguments (i.e., `!eTypeBody.hasLooseBVars`).
  2- The resultant type contains optional/auto params.

  We have considered adding the following extra conditions
    a) The resultant type does not contain any type metavariable.
    b) The resultant type contains a nontype metavariable.

    These two conditions would restrict the method to simple functions that are "morally" in
    the Hindley&Milner fragment.
    If users need to disable expected type propagation, we can add an attribute `[elabWithoutExpectedType]`.
-/
private def propagateExpectedType (arg : Arg) : M Unit := do
  if shouldPropagateExpectedTypeFor arg then
    let s ← get
    -- TODO: handle s.etaArgs.size > 0
    unless !s.etaArgs.isEmpty || !s.propagateExpected do
      match s.expectedType? with
      | none   => pure ()
      | some expectedType =>
        /- We don't propagate `Prop` because we often use `Prop` as a more general "Bool" (e.g., `if-then-else`).
           If we propagate `expectedType == Prop` in the following examples, the elaborator would fail
           ```
           def f1 (s : Nat × Bool) : Bool := if s.2 then false else true

           def f2 (s : List Bool) : Bool := if s.head! then false else true

           def f3 (s : List Bool) : Bool := if List.head! (s.map not) then false else true
           ```
           They would all fail for the same reason. So, let's focus on the first one.
           We would elaborate `s.2` with `expectedType == Prop`.
           Before we elaborate `s`, this method would be invoked, and `s.fType` is `?α × ?β → ?β` and after
           propagation we would have `?α × Prop → Prop`. Then, when we would try to elaborate `s`, and
           get a type error because `?α × Prop` cannot be unified with `Nat × Bool`
           Most users would have a hard time trying to understand why these examples failed.

           Here is a possible alternative workarounds. We give up the idea of using `Prop` at `if-then-else`.
           Drawback: users use `if-then-else` with conditions that are not Decidable.
           So, users would have to embrace `propDecidable` and `choice`.
           This may not be that bad since the developers and users don't seem to care about constructivism.

           We currently use a different workaround, we just don't propagate the expected type when it is `Prop`. -/
        if expectedType.isProp then
```
```

```
          modify fun s => { s with propagateExpected := false }
        else
          let numRemainingArgs := s.args.length
          trace[Elab.app.propagateExpectedType] "etaArgs.size: {s.etaArgs.size}, numRemainingArgs: {numRemainingArgs}, fType: {s.fType}"
          match getForallBody s.explicit numRemainingArgs s.namedArgs s.fType with
          | none           => pure ()
          | some fTypeBody =>
            unless fTypeBody.hasLooseBVars do
              unless (← hasOptAutoParams fTypeBody) do
                trace[Elab.app.propagateExpectedType] "{expectedType} =?= {fTypeBody}"
                if (← isDefEq expectedType fTypeBody) then
                  /- Note that we only set `propagateExpected := false` when propagation has succeeded. -/
                  modify fun s => { s with propagateExpected := false }

/-
  Create a fresh local variable with the current binder name and argument type, add it to `etaArgs` and `f`,
  and then execute the continuation `k`.-/
private def addEtaArg (k : M Expr) : M Expr := do
  let n    ← getBindingName
  let type ← getArgExpectedType
  withLocalDeclD n type fun x => do
    modify fun s => { s with etaArgs := s.etaArgs.push x }
    addNewArg x
    k

/- This method execute after all application arguments have been processed. -/
private def finalize : M Expr := do
  let s ← get
  let mut e := s.f
  -- all user explicit arguments have been consumed
  trace[Elab.app.finalize] e
  let ref ← getRef
  -- Register the error context of implicits
  for mvarId in s.toSetErrorCtx do
    registerMVarErrorImplicitArgInfo mvarId ref e
  if !s.etaArgs.isEmpty then
    e ← mkLambdaFVars s.etaArgs e
  /-
    Remark: we should not use `s.fType` as `eType` even when
    `s.etaArgs.isEmpty`. Reason: it may have been unfolded.
  -/
  let eType ← inferType e
  trace[Elab.app.finalize] "after etaArgs, {e} : {eType}"
  match s.expectedType? with
  | none              => pure ()
  | some expectedType =>
     trace[Elab.app.finalize] "expected type: {expectedType}"
```

```
    -- Try to propagate expected type. Ignore if types are not definitionally equal, caller must handle it.
    discard <| isDefEq expectedType eType
  synthesizeAppInstMVars
  pure e

private def addImplicitArg (k : M Expr) : M Expr := do
  let argType ← getArgExpectedType
  let arg ← mkFreshExprMVar argType
  modify fun s => { s with toSetErrorCtx := s.toSetErrorCtx.push arg.mvarId! }
  addNewArg arg
  k

/- Return true if there is a named argument that depends on the next argument. -/
private def anyNamedArgDependsOnCurrent : M Bool := do
  let s ← get
  if s.namedArgs.isEmpty then
    return false
  else
    forallTelescopeReducing s.fType fun xs _ => do
      let curr := xs[0]
      for i in [1:xs.size] do
        let xDecl ← getLocalDecl xs[i].fvarId!
        if s.namedArgs.any fun arg => arg.name == xDecl.userName then
          if (← getMCtx).localDeclDependsOn xDecl curr.fvarId! then
            return true
      return false

/-
  Process a `fType` of the form `(x : A) → B x`.
  This method assume `fType` is a function type -/
private def processExplictArg (k : M Expr) : M Expr := do
  let s ← get
  match s.args with
  | arg::args =>
    propagateExpectedType arg
    modify fun s => { s with args := args }
    elabAndAddNewArg arg
    k
  | _ =>
    let argType ← getArgExpectedType
    match s.explicit, argType.getOptParamDefault?, argType.getAutoParamTactic? with
    | false, some defVal, _  => addNewArg defVal; k
    | false, _, some (Expr.const tacticDecl _ _) =>
      let env ← getEnv
      let opts ← getOptions
      match evalSyntaxConstant env opts tacticDecl with
      | Except.error err       => throwError err
```

```
      | Except.ok tacticSyntax =>
        -- TODO(Leo): does this work correctly for tactic sequences?
        let tacticBlock ← `(by $tacticSyntax)
        let argType     := argType.getArg! 0 -- `autoParam type := by tactic` ==> `type`
        let argNew := Arg.stx tacticBlock
        propagateExpectedType argNew
        elabAndAddNewArg argNew
        k
    | false, _, some _ =>
      throwError "invalid autoParam, argument must be a constant"
    | _, _, _ =>
      if !s.namedArgs.isEmpty then
        if (← anyNamedArgDependsOnCurrent) then
          addImplicitArg k
        else
          addEtaArg k
      else if !s.explicit then
        if (← fTypeHasOptAutoParams) then
          addEtaArg k
        else if (← get).ellipsis then
          addImplicitArg k
        else
          finalize
      else
        finalize

/-
  Process a `fType` of the form `{x : A} → B x`.
  This method assume `fType` is a function type -/
private def processImplicitArg (k : M Expr) : M Expr := do
  if (← get).explicit then
    processExplictArg k
  else
    addImplicitArg k

/- Return true if the next argument at `args` is of the form `_` -/
private def isNextArgHole : M Bool := do
  match (← get).args with
  | Arg.stx (Syntax.node ``Lean.Parser.Term.hole _) :: _ => pure true
  | _ => pure false

/-
  Process a `fType` of the form `[x : A] → B x`.
  This method assume `fType` is a function type -/
private def processInstImplicitArg (k : M Expr) : M Expr := do
  if (← get).explicit then
    if (← isNextArgHole) then
```

```
      /- Recall that if '@' has been used, and the argument is '_', then we still use type class resolution -/
      let arg ← mkFreshExprMVar (← getArgExpectedType) MetavarKind.synthetic
      modify fun s => { s with args := s.args.tail! }
      addInstMVar arg.mvarId!
      addNewArg arg
      k
    else
      processExplictArg k
  else
    let arg ← mkFreshExprMVar (← getArgExpectedType) MetavarKind.synthetic
    addInstMVar arg.mvarId!
    addNewArg arg
    k

/- Return true if there are regular or named arguments to be processed. -/
private def hasArgsToProcess : M Bool := do
  let s ← get
  pure $ !s.args.isEmpty || !s.namedArgs.isEmpty

/- Elaborate function application arguments. -/
partial def main : M Expr := do
  let s ← get
  let fType ← normalizeFunType
  if fType.isForall then
    let binderName := fType.bindingName!
    let binfo := fType.bindingInfo!
    let s ← get
    match s.namedArgs.find? fun (namedArg : NamedArg) => namedArg.name == binderName with
    | some namedArg =>
      propagateExpectedType namedArg.val
      eraseNamedArg binderName
      elabAndAddNewArg namedArg.val
      main
    | none          =>
      match binfo with
      | BinderInfo.implicit     => processImplicitArg main
      | BinderInfo.instImplicit => processInstImplicitArg main
      | _                       => processExplictArg main
  else if (← hasArgsToProcess) then
    synthesizePendingAndNormalizeFunType
    main
  else
    finalize

end ElabAppArgs

private def propagateExpectedTypeFor (f : Expr) : TermElabM Bool :=
```

```
  match f.getAppFn.constName? with
  | some declName => return !hasElabWithoutExpectedType (← getEnv) declName
  | _ => return true

def elabAppArgs (f : Expr) (namedArgs : Array NamedArg) (args : Array Arg)
    (expectedType? : Option Expr) (explicit ellipsis : Bool) : TermElabM Expr := do
  let fType ← inferType f
  let fType ← instantiateMVars fType
  trace[Elab.app.args] "explicit: {explicit}, {f} : {fType}"
  unless namedArgs.isEmpty && args.isEmpty do
    tryPostponeIfMVar fType
  ElabAppArgs.main.run' {
    args := args.toList,
    expectedType? := expectedType?,
    explicit := explicit,
    ellipsis := ellipsis,
    namedArgs := namedArgs.toList,
    f := f,
    fType := fType
    propagateExpected := (← propagateExpectedTypeFor f)
  }

/-- Auxiliary inductive datatype that represents the resolution of an `LVal`. -/
inductive LValResolution where
  | projFn   (baseStructName : Name) (structName : Name) (fieldName : Name)
  | projIdx  (structName : Name) (idx : Nat)
  | const    (baseStructName : Name) (structName : Name) (constName : Name)
  | localRec (baseName : Name) (fullName : Name) (fvar : Expr)
  | getOp    (fullName : Name) (idx : Syntax)

private def throwLValError {α} (e : Expr) (eType : Expr) (msg : MessageData) : TermElabM α :=
  throwError "{msg}{indentExpr e}\nhas type{indentExpr eType}"

/-- `findMethod? env S fName`.
    1- If `env` contains `S ++ fName`, return `(S, S++fName)`
    2- Otherwise if `env` contains private name `prv` for `S ++ fName`, return `(S, prv)`, o
    3- Otherwise for each parent structure `S'` of  `S`, we try `findMethod? env S' fname` -/
private partial def findMethod? (env : Environment) (structName fieldName : Name) : Option (Name × Name) :=
  let fullName := structName ++ fieldName
  match env.find? fullName with
  | some _ => some (structName, fullName)
  | none   =>
    let fullNamePrv := mkPrivateName env fullName
    match env.find? fullNamePrv with
    | some _ => some (structName, fullNamePrv)
    | none   =>
      if isStructureLike env structName then
```

```
        (getParentStructures env structName).findSome? fun parentStructName => findMethod? env parentStructName fieldName
      else
        none

private def resolveLValAux (e : Expr) (eType : Expr) (lval : LVal) : TermElabM LValResolution := do
  match eType.getAppFn.constName?, lval with
  | some structName, LVal.fieldIdx _ idx =>
    if idx == 0 then
      throwError "invalid projection, index must be greater than 0"
    let env ← getEnv
    unless isStructureLike env structName do
      throwLValError e eType "invalid projection, structure expected"
    let fieldNames := getStructureFields env structName
    if h : idx - 1 < fieldNames.size then
      if isStructure env structName then
        return LValResolution.projFn structName structName (fieldNames.get ⟨idx - 1, h⟩)
      else
        /- `structName` was declared using `inductive` command.
           So, we don't projection functions for it. Thus, we use `Expr.proj` -/
        return LValResolution.projIdx structName (idx - 1)
    else
      throwLValError e eType m!"invalid projection, structure has only {fieldNames.size} field(s)"
  | some structName, LVal.fieldName _ fieldName _ _ =>
    let env ← getEnv
    let searchEnv : Unit → TermElabM LValResolution := fun _ => do
      match findMethod? env structName (Name.mkSimple fieldName) with
      | some (baseStructName, fullName) => pure $ LValResolution.const baseStructName structName fullName
      | none   =>
        throwLValError e eType
          m!"invalid field '{fieldName}', the environment does not contain '{Name.mkStr structName fieldName}'"
    -- search local context first, then environment
    let searchCtx : Unit → TermElabM LValResolution := fun _ => do
      let fullName := Name.mkStr structName fieldName
      let currNamespace ← getCurrNamespace
      let localName := fullName.replacePrefix currNamespace Name.anonymous
      let lctx ← getLCtx
      match lctx.findFromUserName? localName with
      | some localDecl =>
        if localDecl.binderInfo == BinderInfo.auxDecl then
          /- LVal notation is being used to make a "local" recursive call. -/
          pure $ LValResolution.localRec structName fullName localDecl.toExpr
        else
          searchEnv ()
      | none => searchEnv ()
    if isStructure env structName then
      match findField? env structName (Name.mkSimple fieldName) with
      | some baseStructName => pure $ LValResolution.projFn baseStructName structName (Name.mkSimple fieldName)
```

```
      | none                => searchCtx ()
    else
      searchCtx ()
  | some structName, LVal.getOp _ idx =>
    let env ← getEnv
    let fullName := Name.mkStr structName "getOp"
    match env.find? fullName with
    | some _ => pure $ LValResolution.getOp fullName idx
    | none   => throwLValError e eType m!"invalid [..] notation because environment does not contain '{fullName}'"
  | none, LVal.fieldName _ _ (some suffix) _ =>
    if e.isConst then
      throwUnknownConstant (e.constName! ++ suffix)
    else
      throwLValError e eType "invalid field notation, type is not of the form (C ...) where C is a constant"
  | _, LVal.getOp _ idx =>
    throwLValError e eType "invalid [..] notation, type is not of the form (C ...) where C is a constant"
  | _, _ =>
    throwLValError e eType "invalid field notation, type is not of the form (C ...) where C is a constant"

/- whnfCore + implicit consumption.
   Example: given `e` with `eType := {α : Type} → (fun β => List β) α `, it produces `(e ?m, List ?m)` where `?m` is fresh metavariable.
private partial def consumeImplicits (stx : Syntax) (e eType : Expr) : TermElabM (Expr × Expr) := do
  let eType ← whnfCore eType
  match eType with
  | Expr.forallE n d b c =>
    if c.binderInfo.isImplicit then
      let mvar ← mkFreshExprMVar d
      registerMVarErrorHoleInfo mvar.mvarId! stx
      consumeImplicits stx (mkApp e mvar) (b.instantiate1 mvar)
    else if c.binderInfo.isInstImplicit then
      let mvar ← mkInstMVar d
      consumeImplicits stx (mkApp e mvar) (b.instantiate1 mvar)
    else match d.getOptParamDefault? with
      | some defVal => consumeImplicits stx (mkApp e defVal) (b.instantiate1 defVal)
      -- TODO: we do not handle autoParams here.
      | _ => pure (e, eType)
  | _ => pure (e, eType)

private partial def resolveLValLoop (lval : LVal) (e eType : Expr) (previousExceptions : Array Exception) : TermElabM (Expr × LValResolu
  let (e, eType) ← consumeImplicits lval.getRef e eType
  tryPostponeIfMVar eType
  try
    let lvalRes ← resolveLValAux e eType lval
    pure (e, lvalRes)
  catch
    | ex@(Exception.error _ _) =>
      let eType? ← unfoldDefinition? eType
```

```
      match eType? with
      | some eType => resolveLValLoop lval e eType (previousExceptions.push ex)
      | none       =>
        previousExceptions.forM fun ex => logException ex
        throw ex
    | ex@(Exception.internal _ _) => throw ex

private def resolveLVal (e : Expr) (lval : LVal) : TermElabM (Expr × LValResolution) := do
  let eType ← inferType e
  resolveLValLoop lval e eType #[]

private partial def mkBaseProjections (baseStructName : Name) (structName : Name) (e : Expr) : TermElabM Expr := do
  let env ← getEnv
  match getPathToBaseStructure? env baseStructName structName with
  | none => throwError "failed to access field in parent structure"
  | some path =>
    let mut e := e
    for projFunName in path do
      let projFn ← mkConst projFunName
      e ← elabAppArgs projFn #[{ name := `self, val := Arg.expr e }] (args := #[]) (expectedType? := none) (explicit := false) (ellipsis
    return e

/- Auxiliary method for field notation. It tries to add `e` as a new argument to `args` or `namedArgs`.
   This method first finds the parameter with a type of the form `(baseName ...)`.
   When the parameter is found, if it an explicit one and `args` is big enough, we add `e` to `args`.
   Otherwise, if there isn't another parameter with the same name, we add `e` to `namedArgs`.

   Remark: `fullName` is the name of the resolved "field" access function. It is used for reporting errors -/
private def addLValArg (baseName : Name) (fullName : Name) (e : Expr) (args : Array Arg) (namedArgs : Array NamedArg) (fType : Expr)
    : TermElabM (Array Arg × Array NamedArg) :=
  forallTelescopeReducing fType fun xs _ => do
    let mut argIdx := 0 -- position of the next explicit argument
    let mut remainingNamedArgs := namedArgs
    for i in [:xs.size] do
      let x := xs[i]
      let xDecl ← getLocalDecl x.fvarId!
      /- If there is named argument with name `xDecl.userName`, then we skip it. -/
      match remainingNamedArgs.findIdx? (fun namedArg => namedArg.name == xDecl.userName) with
      | some idx =>
        remainingNamedArgs := remainingNamedArgs.eraseIdx idx
      | none =>
        let mut foundIt := false
        let type := xDecl.type
        if type.consumeMData.isAppOf baseName then
          foundIt := true
        if !foundIt then
          /- Normalize type and try again -/
```

```
        let type ← withReducible $ whnf type
        if type.consumeMData.isAppOf baseName then
          foundIt := true
      if foundIt then
        /- We found a type of the form (baseName ...).
           First, we check if the current argument is an explicit one,
           and the current explicit position "fits" at `args` (i.e., it must be ≤ arg.size) -/
        if argIdx ≤ args.size && xDecl.binderInfo.isExplicit then
          /- We insert `e` as an explicit argument -/
          return (args.insertAt argIdx (Arg.expr e), namedArgs)
        /- If we can't add `e` to `args`, we try to add it using a named argument, but this is only possible
           if there isn't an argument with the same name occurring before it. -/
        for j in [:i] do
          let prev := xs[j]
          let prevDecl ← getLocalDecl prev.fvarId!
          if prevDecl.userName == xDecl.userName then
            throwError "invalid field notation, function '{fullName}' has argument with the expected type{indentExpr type}\nbut it can
        return (args, namedArgs.push { name := xDecl.userName, val := Arg.expr e })
      if xDecl.binderInfo.isExplicit then
        -- advance explicit argument position
        argIdx := argIdx + 1
  throwError "invalid field notation, function '{fullName}' does not have argument with type ({baseName} ...) that can be used, it mus

private def elabAppLValsAux (namedArgs : Array NamedArg) (args : Array Arg) (expectedType? : Option Expr) (explicit ellipsis : Bool)
    (f : Expr) (lvals : List LVal) : TermElabM Expr :=
  let rec loop : Expr → List LVal → TermElabM Expr
  | f, []          => elabAppArgs f namedArgs args expectedType? explicit ellipsis
  | f, lval::lvals => do
    if let LVal.fieldName (ref := fieldStx) (targetStx := targetStx) .. := lval then
      addDotCompletionInfo  targetStx f expectedType? fieldStx
    let (f, lvalRes) ← resolveLVal f lval
    match lvalRes with
    | LValResolution.projIdx structName idx =>
      let f := mkProj structName idx f
      addTermInfo lval.getRef f
      loop f lvals
    | LValResolution.projFn baseStructName structName fieldName =>
      let f ← mkBaseProjections baseStructName structName f
      let projFn ← mkConst (baseStructName ++ fieldName)
      addTermInfo lval.getRef projFn
      if lvals.isEmpty then
        let namedArgs ← addNamedArg namedArgs { name := `self, val := Arg.expr f }
        elabAppArgs projFn namedArgs args expectedType? explicit ellipsis
      else
        let f ← elabAppArgs projFn #[{ name := `self, val := Arg.expr f }] #[] (expectedType? := none) (explicit := false) (ellipsis :=
        loop f lvals
    | LValResolution.const baseStructName structName constName =>
```

```
      let f ← if baseStructName != structName then mkBaseProjections baseStructName structName f else pure f
      let projFn ← mkConst constName
      addTermInfo lval.getRef projFn
      if lvals.isEmpty then
        let projFnType ← inferType projFn
        let (args, namedArgs) ← addLValArg baseStructName constName f args namedArgs projFnType
        elabAppArgs projFn namedArgs args expectedType? explicit ellipsis
      else
        let f ← elabAppArgs projFn #[] #[Arg.expr f] (expectedType? := none) (explicit := false) (ellipsis := false)
        loop f lvals
    | LValResolution.localRec baseName fullName fvar =>
      addTermInfo lval.getRef fvar
      if lvals.isEmpty then
        let fvarType ← inferType fvar
        let (args, namedArgs) ← addLValArg baseName fullName f args namedArgs fvarType
        elabAppArgs fvar namedArgs args expectedType? explicit ellipsis
      else
        let f ← elabAppArgs fvar #[] #[Arg.expr f] (expectedType? := none) (explicit := false) (ellipsis := false)
        loop f lvals
    | LValResolution.getOp fullName idx =>
      let getOpFn ← mkConst fullName
      addTermInfo lval.getRef getOpFn
      if lvals.isEmpty then
        let namedArgs ← addNamedArg namedArgs { name := `self, val := Arg.expr f }
        let namedArgs ← addNamedArg namedArgs { name := `idx,  val := Arg.stx idx }
        elabAppArgs getOpFn namedArgs args expectedType? explicit ellipsis
      else
        let f ← elabAppArgs getOpFn #[{ name := `self, val := Arg.expr f }, { name := `idx, val := Arg.stx idx }]
                            #[] (expectedType? := none) (explicit := false) (ellipsis := false)
        loop f lvals
  loop f lvals

private def elabAppLVals (f : Expr) (lvals : List LVal) (namedArgs : Array NamedArg) (args : Array Arg)
    (expectedType? : Option Expr) (explicit ellipsis : Bool) : TermElabM Expr := do
  if !lvals.isEmpty && explicit then
    throwError "invalid use of field notation with `@` modifier"
  elabAppLValsAux namedArgs args expectedType? explicit ellipsis f lvals

def elabExplicitUnivs (lvls : Array Syntax) : TermElabM (List Level) := do
  lvls.foldrM (fun stx lvls => do pure ((← elabLevel stx)::lvls)) []

/-
Interaction between `errToSorry` and `observing`.

- The method `elabTerm` catches exceptions, log them, and returns a synthetic sorry (IF `ctx.errToSorry` == true).

- When we elaborate choice nodes (and overloaded identifiers), we track multiple results using the `observing x` combinator.
```

```
  The `observing x` executes `x` and returns a `TermElabResult`.

`observing `x does not check for synthetic sorry's, just an exception. Thus, it may think `x` worked when it didn't
if a synthetic sorry was introduced. We decided that checking for synthetic sorrys at `observing` is not a good solution
because it would not be clear to decide what the "main" error message for the alternative is. When the result contains
a synthetic `sorry`, it is not clear which error message corresponds to the `sorry`. Moreover, while executing `x`, many
error messages may have been logged. Recall that we need an error per alternative at `mergeFailures`.

Thus, we decided to set `errToSorry` to `false` whenever processing choice nodes and overloaded symbols.

Important: we rely on the property that after `errToSorry` is set to
false, no elaboration function executed by `x` will reset it to
`true`.
-/

private partial def elabAppFnId (fIdent : Syntax) (fExplicitUnivs : List Level) (lvals : List LVal)
    (namedArgs : Array NamedArg) (args : Array Arg) (expectedType? : Option Expr) (explicit ellipsis overloaded : Bool) (acc : Array (Te
    : TermElabM (Array (TermElabResult Expr)) := do
  let funLVals ← withRef fIdent <| resolveName' fIdent fExplicitUnivs expectedType?
  let overloaded := overloaded || funLVals.length > 1
  -- Set `errToSorry` to `false` if `funLVals` > 1. See comment above about the interaction between `errToSorry` and `observing`.
  withReader (fun ctx => { ctx with errToSorry := funLVals.length == 1 && ctx.errToSorry }) do
    funLVals.foldlM (init := acc) fun acc (f, fIdent, fields) => do
      addTermInfo fIdent f
      let lvals' := toLVals fields (first := true)
      let s ← observing do
        let e ← elabAppLVals f (lvals' ++ lvals) namedArgs args expectedType? explicit ellipsis
        if overloaded then ensureHasType expectedType? e else pure e
      return acc.push s
where
  toName : List Syntax → Name
    | []              => Name.anonymous
    | field :: fields => Name.mkStr (toName fields) field.getId.toString

  toLVals : List Syntax → (first : Bool) → List LVal
    | [],            _      => []
    | field::fields, true  => LVal.fieldName field field.getId.toString (toName (field::fields)) fIdent :: toLVals fields false
    | field::fields, false => LVal.fieldName field field.getId.toString none fIdent :: toLVals fields false

private partial def elabAppFn (f : Syntax) (lvals : List LVal) (namedArgs : Array NamedArg) (args : Array Arg)
    (expectedType? : Option Expr) (explicit ellipsis overloaded : Bool) (acc : Array (TermElabResult Expr)) : TermElabM (Array (TermElabI
  if f.getKind == choiceKind then
    -- Set `errToSorry` to `false` when processing choice nodes. See comment above about the interaction between `errToSorry` and `obser
    withReader (fun ctx => { ctx with errToSorry := false }) do
      f.getArgs.foldlM (fun acc f => elabAppFn f lvals namedArgs args expectedType? explicit ellipsis true acc) acc
  else
    let elabFieldName (e field : Syntax) := do
```

```
      let newLVals := field.getId.eraseMacroScopes.components.map fun n =>
        -- We use `none` here since `field` can't be part of a composite name
        LVal.fieldName field (toString n) none e
      elabAppFn e (newLVals ++ lvals) namedArgs args expectedType? explicit ellipsis overloaded acc
    let elabFieldIdx (e idxStx : Syntax) := do
      let idx := idxStx.isFieldIdx?.get!
      elabAppFn e (LVal.fieldIdx idxStx idx :: lvals) namedArgs args expectedType? explicit ellipsis overloaded acc
    match f with
    | `($(e).$idx:fieldIdx) => elabFieldIdx e idx
    | `($e |>.$idx:fieldIdx) => elabFieldIdx e idx
    | `($(e).$field:ident) => elabFieldName e field
    | `($e |>.$field:ident) => elabFieldName e field
    | `($e[%$bracket $idx]) => elabAppFn e (LVal.getOp bracket idx :: lvals) namedArgs args expectedType? explicit ellipsis overloaded a
    | `($id:ident@$t:term) =>
      throwError "unexpected occurrence of named pattern"
    | `($id:ident) => do
      elabAppFnId id [] lvals namedArgs args expectedType? explicit ellipsis overloaded acc
    | `($id:ident.{$us,*}) => do
      let us ← elabExplicitUnivs us
      elabAppFnId id us lvals namedArgs args expectedType? explicit ellipsis overloaded acc
    | `(@$id:ident) =>
      elabAppFn id lvals namedArgs args expectedType? (explicit := true) ellipsis overloaded acc
    | `(@$id:ident.{$us,*}) =>
      elabAppFn (f.getArg 1) lvals namedArgs args expectedType? (explicit := true) ellipsis overloaded acc
    | `(@$t)     => throwUnsupportedSyntax -- invalid occurrence of `@`
    | `(_)       => throwError "placeholders '_' cannot be used where a function is expected"
    | _ => do
      let catchPostpone := !overloaded
      /- If we are processing a choice node, then we should use `catchPostpone == false` when elaborating terms.
        Recall that `observing` does not catch `postponeExceptionId`. -/
      if lvals.isEmpty && namedArgs.isEmpty && args.isEmpty then
        /- Recall that elabAppFn is used for elaborating atomics terms **and** choice nodes that may contain
          arbitrary terms. If they are not being used as a function, we should elaborate using the expectedType. -/
        let s ←
          if overloaded then
            observing <| elabTermEnsuringType f expectedType? catchPostpone
          else
            observing <| elabTerm f expectedType?
        return acc.push s
      else
        let s ← observing do
          let f ← elabTerm f none catchPostpone
          let e ← elabAppLVals f lvals namedArgs args expectedType? explicit ellipsis
          if overloaded then ensureHasType expectedType? e else pure e
        return acc.push s

private def isSuccess (candidate : TermElabResult Expr) : Bool :=
```

```
  match candidate with
  | EStateM.Result.ok _ _ => true
  | _ => false

private def getSuccess (candidates : Array (TermElabResult Expr)) : Array (TermElabResult Expr) :=
  candidates.filter isSuccess

private def toMessageData (ex : Exception) : TermElabM MessageData := do
  let pos ← getRefPos
  match ex.getRef.getPos? with
  | none       => return ex.toMessageData
  | some exPos =>
    if pos == exPos then
      return ex.toMessageData
    else
      let exPosition := (← getFileMap).toPosition exPos
      return m!"{exPosition.line}:{exPosition.column} {ex.toMessageData}"

private def toMessageList (msgs : Array MessageData) : MessageData :=
  indentD (MessageData.joinSep msgs.toList m!"\n\n")

private def mergeFailures {α} (failures : Array (TermElabResult Expr)) : TermElabM α := do
  let msgs ← failures.mapM fun failure =>
    match failure with
    | EStateM.Result.ok _ _     => unreachable!
    | EStateM.Result.error ex _ => toMessageData ex
  throwError "overloaded, errors {toMessageList msgs}"

private def elabAppAux (f : Syntax) (namedArgs : Array NamedArg) (args : Array Arg) (ellipsis : Bool) (expectedType? : Option Expr) : Te
  let candidates ← elabAppFn f [] namedArgs args expectedType? (explicit := false) (ellipsis := ellipsis) (overloaded := false) #[]
  if candidates.size == 1 then
    applyResult candidates[0]
  else
    let successes := getSuccess candidates
    if successes.size == 1 then
      applyResult successes[0]
    else if successes.size > 1 then
      let lctx ← getLCtx
      let opts ← getOptions
      let msgs : Array MessageData := successes.map fun success => match success with
        | EStateM.Result.ok e s => MessageData.withContext { env := s.meta.core.env, mctx := s.meta.meta.mctx, lctx := lctx, opts := opt
        | _                     => unreachable!
      throwErrorAt f "ambiguous, possible interpretations {toMessageList msgs}"
    else
      withRef f <| mergeFailures candidates

partial def expandArgs (args : Array Syntax) (pattern := false) : TermElabM (Array NamedArg × Array Arg × Bool) := do
```

```
  let (args, ellipsis) :=
    if args.isEmpty then
      (args, false)
    else if args.back.isOfKind ``Lean.Parser.Term.ellipsis then
      (args.pop, true)
    else
      (args, false)
  let (namedArgs, args) ← args.foldlM (init := (#[], #[])) fun (namedArgs, args) stx => do
    if stx.getKind == ``Lean.Parser.Term.namedArgument then
      -- trailing_tparser try ("(" >> ident >> " := ") >> termParser >> ")"
      let name := stx[1].getId.eraseMacroScopes
      let val  := stx[3]
      let namedArgs ← addNamedArg namedArgs { ref := stx, name := name, val := Arg.stx val }
      return (namedArgs, args)
    else if stx.getKind == ``Lean.Parser.Term.ellipsis then
      throwErrorAt stx "unexpected '..'"
    else
      return (namedArgs, args.push $ Arg.stx stx)
  return (namedArgs, args, ellipsis)

def expandApp (stx : Syntax) (pattern := false) : TermElabM (Syntax × Array NamedArg × Array Arg × Bool) := do
  let (namedArgs, args, ellipsis) ← expandArgs stx[1].getArgs
  return (stx[0], namedArgs, args, ellipsis)

@[builtinTermElab app] def elabApp : TermElab := fun stx expectedType? =>
  withoutPostponingUniverseConstraints do
    let (f, namedArgs, args, ellipsis) ← expandApp stx
    elabAppAux f namedArgs args (ellipsis := ellipsis) expectedType?

private def elabAtom : TermElab := fun stx expectedType? =>
  elabAppAux stx #[] #[] (ellipsis := false) expectedType?

@[builtinTermElab ident] def elabIdent : TermElab := elabAtom
@[builtinTermElab namedPattern] def elabNamedPattern : TermElab := elabAtom
@[builtinTermElab explicitUniv] def elabExplicitUniv : TermElab := elabAtom
@[builtinTermElab pipeProj] def elabPipeProj : TermElab
  | `($e |>.$f $args*), expectedType? =>
    withoutPostponingUniverseConstraints do
      let (namedArgs, args, ellipsis) ← expandArgs args
      elabAppAux (← `($e |>.$f)) namedArgs args (ellipsis := ellipsis) expectedType?
  | _, _ => throwUnsupportedSyntax

@[builtinTermElab explicit] def elabExplicit : TermElab := fun stx expectedType? =>
  match stx with
  | `(@$id:ident)         => elabAtom stx expectedType?  -- Recall that `elabApp` also has support for `@`
  | `(@$id:ident.{$us,*}) => elabAtom stx expectedType?
  | `(@($t))              => elabTerm t expectedType? (implicitLambda := false)    -- `@` is being used just to disable implicit lambdas
```

```
  | `(@$t)                 => elabTerm t expectedType? (implicitLambda := false)   -- `@` is being used just to disable implicit lambdas
  | _                      => throwUnsupportedSyntax

@[builtinTermElab choice] def elabChoice : TermElab := elabAtom
@[builtinTermElab proj] def elabProj : TermElab := elabAtom
@[builtinTermElab arrayRef] def elabArrayRef : TermElab := elabAtom

builtin_initialize
  registerTraceClass `Elab.app

end Lean.Elab.Term
::::::::::::::
Elab/Attributes.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Attr
import Lean.Attributes
import Lean.MonadEnv
import Lean.Elab.Util
namespace Lean.Elab

structure Attribute where
  kind  : AttributeKind := AttributeKind.global
  name  : Name
  stx   : Syntax := Syntax.missing

instance : ToFormat Attribute where
  format attr :=
   let kindStr := match attr.kind with
     | AttributeKind.global => ""
     | AttributeKind.local  => "local "
     | AttributeKind.scoped => "scoped "
   Format.bracket "@[" f!"{kindStr}{attr.name}{toString attr.stx}" "]"

instance : Inhabited Attribute where
  default := { name := arbitrary }

/-
  ```
  attrKind := leading_parser optional («scoped» <|> «local»)
  ```
-/
def toAttributeKind [Monad m] [MonadResolveName m] [MonadError m] (attrKindStx : Syntax) : m AttributeKind := do
```

```
  if attrKindStx[0].isNone then
    return AttributeKind.global
  else if attrKindStx[0][0].getKind == ``Lean.Parser.Term.scoped then
    if (← getCurrNamespace).isAnonymous then
      throwError "scoped attributes must be used inside namespaces"
    return AttributeKind.scoped
  else
    return AttributeKind.local

def mkAttrKindGlobal : Syntax :=
  Syntax.node ``Lean.Parser.Term.attrKind #[mkNullNode]

def elabAttr {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (attrInstance : Synt
  /- attrInstance     := ppGroup $ leading_parser attrKind >> attrParser -/
  let attrKind ← toAttributeKind attrInstance[0]
  let attr := attrInstance[1]
  let attr ← liftMacroM <| expandMacros attr
  let attrName ←
    if attr.getKind == ``Parser.Attr.simple then
      pure attr[0].getId.eraseMacroScopes
    else
      match attr.getKind with
      | Name.str _ s _ => pure <| Name.mkSimple s
      | _ => throwErrorAt attr  "unknown attribute"
  unless isAttribute (← getEnv) attrName do
    throwError "unknown attribute [{attrName}]"
  /- The `AttrM` does not have sufficient information for expanding macros in `args`.
     So, we expand them before here before we invoke the attributer handlers implemented using `AttrM`. -/
  pure { kind := attrKind, name := attrName, stx := attr }

def elabAttrs {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (attrInstances : Ar
  let mut attrs := #[]
  for attr in attrInstances do
    attrs := attrs.push (← elabAttr attr)
  return attrs

-- leading_parser "@[" >> sepBy1 attrInstance ", " >> "]"
def elabDeclAttrs {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (stx : Syntax)
  elabAttrs stx[1].getSepArgs

end Lean.Elab
::::::::::::::
Elab/AutoBound.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
```

```
Authors: Leonardo de Moura
-/
import Lean.Data.Options

/- Basic support for auto bound implicit local names -/

namespace Lean.Elab

register_builtin_option autoBoundImplicitLocal : Bool := {
    defValue := true
    descr    := "Unbound local variables in declaration headers become implicit arguments if they are a lower case or greek letter follo
  }

private def isValidAutoBoundSuffix (s : String) : Bool :=
  s.toSubstring.drop 1 |>.all fun c => c.isDigit || isSubScriptAlnum c || c == '_' || c == '\''

/-
Remark: Issue #255 exposed a nasty interaction between macro scopes and auto-bound-implicit names.
```
local notation "A" => id x
theorem test : A = A := sorry
```
We used to use `n.eraseMacroScopes` at `isValidAutoBoundImplicitName` and `isValidAutoBoundLevelName`.
Thus, in the example above, when `A` is expanded, a `x` with a fresh macro scope is created.
`x`+macros-scope is not in scope and is a valid auto-bound implicit name after macro scopes are erased.
So, an auto-bound exception would be thrown, and `x`+macro-scope would be added as a new implicit.
When, we try again, a `x` with a new macro scope is created and this process keeps repeating.
Therefore, we do consider identifier with macro scopes anymore.
-/

def isValidAutoBoundImplicitName (n : Name) : Bool :=
  match n with
  | Name.str Name.anonymous s _ => s.length > 0 && (isGreek s[0] || s[0].isLower) && isValidAutoBoundSuffix s
  | _ => false

def isValidAutoBoundLevelName (n : Name) : Bool :=
  match n with
  | Name.str Name.anonymous s _ => s.length > 0 && s[0].isLower && isValidAutoBoundSuffix s
  | _ => false

end Lean.Elab
::::::::::::::
Elab/Binders.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
```

```
Authors: Leonardo de Moura
-/
import Lean.Elab.Term
import Lean.Parser.Term

namespace Lean.Elab.Term
open Meta
open Lean.Parser.Term

/--
  Given syntax of the forms
    a) (`:` term)?
    b) `:` term
  return `term` if it is present, or a hole if not. -/
private def expandBinderType (ref : Syntax) (stx : Syntax) : Syntax :=
  if stx.getNumArgs == 0 then
    mkHole ref
  else
    stx[1]

/-- Given syntax of the form `ident <|> hole`, return `ident`. If `hole`, then we create a new anonymous name. -/
private def expandBinderIdent (stx : Syntax) : TermElabM Syntax :=
  match stx with
  | `(_) => mkFreshIdent stx
  | _    => pure stx

/-- Given syntax of the form `(ident >> " : ")?`, return `ident`, or a new instance name. -/
private def expandOptIdent (stx : Syntax) : TermElabM Syntax := do
  if stx.isNone then
    let id ← withFreshMacroScope <| MonadQuotation.addMacroScope `inst
    return mkIdentFrom stx id
  else
    return stx[0]

structure BinderView where
  id : Syntax
  type : Syntax
  bi : BinderInfo

partial def quoteAutoTactic : Syntax → TermElabM Syntax
  | stx@(Syntax.ident _ _ _ _) => throwErrorAt stx "invalid auto tactic, identifier is not allowed"
  | stx@(Syntax.node k args)   => do
    if stx.isAntiquot then
      throwErrorAt stx "invalid auto tactic, antiquotation is not allowed"
    else
      let mut quotedArgs ← `(Array.empty)
      for arg in args do
```

```lean
      if k == nullKind && (arg.isAntiquotSuffixSplice || arg.isAntiquotSplice) then
        throwErrorAt arg "invalid auto tactic, antiquotation is not allowed"
      else
        let quotedArg ← quoteAutoTactic arg
        quotedArgs ← `(Array.push $quotedArgs $quotedArg)
    `(Syntax.node $(quote k) $quotedArgs)
  | Syntax.atom info val => `(mkAtom $(quote val))
  | Syntax.missing       => unreachable!

def declareTacticSyntax (tactic : Syntax) : TermElabM Name :=
  withFreshMacroScope do
    let name ← MonadQuotation.addMacroScope `_auto
    let type := Lean.mkConst `Lean.Syntax
    let tactic ← quoteAutoTactic tactic
    let val ← elabTerm tactic type
    let val ← instantiateMVars val
    trace[Elab.autoParam] val
    let decl := Declaration.defnDecl { name := name, levelParams := [], type := type, value := val, hints := ReducibilityHints.opaque,
                                       safety := DefinitionSafety.safe }
    addDecl decl
    compileDecl decl
    return name

/-
Expand `optional (binderTactic <|> binderDefault)`
def binderTactic  := leading_parser " := " >> " by " >> tacticParser
def binderDefault := leading_parser " := " >> termParser
-/
private def expandBinderModifier (type : Syntax) (optBinderModifier : Syntax) : TermElabM Syntax := do
  if optBinderModifier.isNone then
    return type
  else
    let modifier := optBinderModifier[0]
    let kind     := modifier.getKind
    if kind == `Lean.Parser.Term.binderDefault then
      let defaultVal := modifier[1]
      `(optParam $type $defaultVal)
    else if kind == `Lean.Parser.Term.binderTactic then
      let tac := modifier[2]
      let name ← declareTacticSyntax tac
      `(autoParam $type $(mkIdentFrom tac name))
    else
      throwUnsupportedSyntax

private def getBinderIds (ids : Syntax) : TermElabM (Array Syntax) :=
  ids.getArgs.mapM fun id =>
    let k := id.getKind
```

```
    if k == identKind || k == `Lean.Parser.Term.hole then
      return id
    else
      throwErrorAt id "identifier or `_` expected"

/-
  Recall that
  ```
  def typeSpec := leading_parser " : " >> termParser
  def optType : Parser := optional typeSpec
  ```
-/
def expandOptType (ref : Syntax) (optType : Syntax) : Syntax :=
  if optType.isNone then
    mkHole ref
  else
    optType[0][1]

private def matchBinder (stx : Syntax) : TermElabM (Array BinderView) := do
  let k := stx.getKind
  if k == `Lean.Parser.Term.simpleBinder then
    -- binderIdent+ >> optType
    let ids ← getBinderIds stx[0]
    let type := expandOptType stx stx[1]
    ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.default }
  else if k == `Lean.Parser.Term.explicitBinder then
    -- `(` binderIdent+ binderType (binderDefault <|> binderTactic)? `)`
    let ids ← getBinderIds stx[1]
    let type        := expandBinderType stx stx[2]
    let optModifier := stx[3]
    let type ← expandBinderModifier type optModifier
    ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.default }
  else if k == `Lean.Parser.Term.implicitBinder then
    -- `{` binderIdent+ binderType `}`
    let ids ← getBinderIds stx[1]
    let type := expandBinderType stx stx[2]
    ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.implicit }
  else if k == `Lean.Parser.Term.instBinder then
    -- `[` optIdent type `]`
    let id ← expandOptIdent stx[1]
    let type := stx[2]
    pure #[ { id := id, type := type, bi := BinderInfo.instImplicit } ]
  else
    throwUnsupportedSyntax

private def registerFailedToInferBinderTypeInfo (type : Expr) (ref : Syntax) : TermElabM Unit :=
  registerCustomErrorIfMVar type ref "failed to infer binder type"
```

```
private def addLocalVarInfoCore (lctx : LocalContext) (stx : Syntax) (fvar : Expr) : TermElabM Unit := do
  if (← getInfoState).enabled then
    pushInfoTree <| InfoTree.node (children := {}) <| Info.ofTermInfo { lctx := lctx, expr := fvar, stx := stx }

private def addLocalVarInfo (stx : Syntax) (fvar : Expr) : TermElabM Unit := do
  addLocalVarInfoCore (← getLCtx) stx fvar

private def ensureAtomicBinderName (binderView : BinderView) : TermElabM Unit :=
  let n := binderView.id.getId.eraseMacroScopes
  unless n.isAtomic do
    throwErrorAt binderView.id "invalid binder name '{n}', it must be atomic"

private partial def elabBinderViews {α} (binderViews : Array BinderView) (fvars : Array Expr) (k : Array Expr → TermElabM α)
    : TermElabM α :=
  let rec loop (i : Nat) (fvars : Array Expr) : TermElabM α := do
    if h : i < binderViews.size then
      let binderView := binderViews.get ⟨i, h⟩
      ensureAtomicBinderName binderView
      let type ← elabType binderView.type
      registerFailedToInferBinderTypeInfo type binderView.type
      withLocalDecl binderView.id.getId binderView.bi type fun fvar => do
        addLocalVarInfo binderView.id fvar
        loop (i+1) (fvars.push fvar)
    else
      k fvars
  loop 0 fvars

private partial def elabBindersAux {α} (binders : Array Syntax) (k : Array Expr → TermElabM α) : TermElabM α :=
  let rec loop (i : Nat) (fvars : Array Expr) : TermElabM α := do
    if h : i < binders.size then
      let binderViews ← matchBinder (binders.get ⟨i, h⟩)
      elabBinderViews binderViews fvars <| loop (i+1)
    else
      k fvars
  loop 0 #[]

/--
  Elaborate the given binders (i.e., `Syntax` objects for `simpleBinder <|> bracketedBinder`),
  update the local context, set of local instances, reset instance chache (if needed), and then
  execute `x` with the updated context. -/
def elabBinders {α} (binders : Array Syntax) (k : Array Expr → TermElabM α) : TermElabM α :=
  withoutPostponingUniverseConstraints do
    if binders.isEmpty then
      k #[]
    else
      elabBindersAux binders k
```

```
@[inline] def elabBinder {α} (binder : Syntax) (x : Expr → TermElabM α) : TermElabM α :=
  elabBinders #[binder] fun fvars => x fvars[0]

@[builtinTermElab «forall»] def elabForall : TermElab := fun stx _ =>
  match stx with
  | `(forall $binders*, $term) =>
    elabBinders binders fun xs => do
      let e ← elabType term
      mkForallFVars xs e
  | _ => throwUnsupportedSyntax

@[builtinTermElab arrow] def elabArrow : TermElab :=
  adaptExpander fun stx => match stx with
  | `($dom:term -> $rng) => `(forall (a : $dom), $rng)
  | _                    => throwUnsupportedSyntax

@[builtinTermElab depArrow] def elabDepArrow : TermElab := fun stx _ =>
  -- bracketedBinder `->` term
  let binder := stx[0]
  let term   := stx[2]
  elabBinders #[binder] fun xs => do
    mkForallFVars xs (← elabType term)

/--
  Auxiliary functions for converting `id_1 ... id_n` application into `#[id_1, ..., id_m]`
  It is used at `expandFunBinders`. -/
private partial def getFunBinderIds? (stx : Syntax) : OptionT TermElabM (Array Syntax) :=
  let convertElem (stx : Syntax) : OptionT TermElabM Syntax :=
    match stx with
    | `(_) => do let ident ← mkFreshIdent stx; pure ident
    | `($id:ident) => return id
    | _ => failure
  match stx with
  | `($f $args*) => do
     let mut acc := #[].push (← convertElem f)
     for arg in args do
       acc := acc.push (← convertElem arg)
     return acc
  | _ =>
    return #[].push (← convertElem stx)

/--
  Auxiliary function for expanding `fun` notation binders. Recall that `fun` parser is defined as
  ```
  def funBinder : Parser := implicitBinder <|> instBinder <|> termParser maxPrec
  leading_parser unicodeSymbol "λ" "fun" >> many1 funBinder >> "=>" >> termParser
```
```

```
  ```
  to allow notation such as `fun (a, b) => a + b`, where `(a, b)` should be treated as a pattern.
  The result is a pair `(explicitBinders, newBody)`, where `explicitBinders` is syntax of the form
  ```
  `(` ident `:` term `)`
  ```
  which can be elaborated using `elabBinders`, and `newBody` is the updated `body` syntax.
  We update the `body` syntax when expanding the pattern notation.
  Example: `fun (a, b) => a + b` expands into `fun _a_1 => match _a_1 with | (a, b) => a + b`.
  See local function `processAsPattern` at `expandFunBindersAux`.

  The resulting `Bool` is true if a pattern was found. We use it "mark" a macro expansion. -/
partial def expandFunBinders (binders : Array Syntax) (body : Syntax) : TermElabM (Array Syntax × Syntax × Bool) :=
  let rec loop (body : Syntax) (i : Nat) (newBinders : Array Syntax) := do
    if h : i < binders.size then
      let binder := binders.get ⟨i, h⟩
      let processAsPattern : Unit → TermElabM (Array Syntax × Syntax × Bool) := fun _ => do
        let pattern := binder
        let major ← mkFreshIdent binder
        let (binders, newBody, _) ← loop body (i+1) (newBinders.push $ mkExplicitBinder major (mkHole binder))
        let newBody ← `(match $major:ident with | $pattern => $newBody)
        pure (binders, newBody, true)
      match binder with
      | Syntax.node `Lean.Parser.Term.implicitBinder _ => loop body (i+1) (newBinders.push binder)
      | Syntax.node `Lean.Parser.Term.instBinder _   => loop body (i+1) (newBinders.push binder)
      | Syntax.node `Lean.Parser.Term.explicitBinder _ => loop body (i+1) (newBinders.push binder)
      | Syntax.node `Lean.Parser.Term.simpleBinder _   => loop body (i+1) (newBinders.push binder)
      | Syntax.node `Lean.Parser.Term.hole _ =>
        let ident ← mkFreshIdent binder
        let type := binder
        loop body (i+1) (newBinders.push <| mkExplicitBinder ident type)
      | Syntax.node `Lean.Parser.Term.paren args =>
        -- `(` (termParser >> parenSpecial)? `)`
        -- parenSpecial := (tupleTail <|> typeAscription)?
        let binderBody := binder[1]
        if binderBody.isNone then
          processAsPattern ()
        else
          let idents := binderBody[0]
          let special := binderBody[1]
          if special.isNone then
            processAsPattern ()
          else if special[0].getKind != `Lean.Parser.Term.typeAscription then
            processAsPattern ()
          else
            -- typeAscription := `:` term
            let type := special[0][1]
```
```

```
          match (← getFunBinderIds? idents) with
          | some idents => loop body (i+1) (newBinders ++ idents.map (fun ident => mkExplicitBinder ident type))
          | none        => processAsPattern ()
      | Syntax.ident .. =>
        let type  := mkHole binder
        loop body (i+1) (newBinders.push <| mkExplicitBinder binder type)
      | _ => processAsPattern ()
    else
      pure (newBinders, body, false)
  loop body 0 #[]

namespace FunBinders

structure State where
  fvars         : Array Expr := #[]
  lctx          : LocalContext
  localInsts    : LocalInstances
  expectedType? : Option Expr := none

private def propagateExpectedType (fvar : Expr) (fvarType : Expr) (s : State) : TermElabM State := do
  match s.expectedType? with
  | none              => pure s
  | some expectedType =>
    let expectedType ← whnfForall expectedType
    match expectedType with
    | Expr.forallE _ d b _ =>
      discard <| isDefEq fvarType d
      let b := b.instantiate1 fvar
      pure { s with expectedType? := some b }
    | _ => pure { s with expectedType? := none }

private partial def elabFunBinderViews (binderViews : Array BinderView) (i : Nat) (s : State) : TermElabM State := do
  if h : i < binderViews.size then
    let binderView := binderViews.get ⟨i, h⟩
    ensureAtomicBinderName binderView
    withRef binderView.type <| withLCtx s.lctx s.localInsts do
      let type       ← elabType binderView.type
      registerFailedToInferBinderTypeInfo type binderView.type
      let fvarId ← mkFreshFVarId
      let fvar  := mkFVar fvarId
      let s     := { s with fvars := s.fvars.push fvar }
      -- dbgTrace (toString binderView.id.getId ++ " : " ++ toString type)
      /-
        We do **not** want to support default and auto arguments in lambda abstractions.
        Example: `fun (x : Nat := 10) => x+1`.
        We do not believe this is an useful feature, and it would complicate the logic here.
      -/
```

```
      let lctx  := s.lctx.mkLocalDecl fvarId binderView.id.getId type binderView.bi
      addLocalVarInfoCore lctx binderView.id fvar
      let s ← withRef binderView.id <| propagateExpectedType fvar type s
      let s := { s with lctx := lctx }
      match (← isClass? type) with
      | none           => elabFunBinderViews binderViews (i+1) s
      | some className =>
        resettingSynthInstanceCache do
          let localInsts := s.localInsts.push { className := className, fvar := mkFVar fvarId }
          elabFunBinderViews binderViews (i+1) { s with localInsts := localInsts }
  else
    pure s

partial def elabFunBindersAux (binders : Array Syntax) (i : Nat) (s : State) : TermElabM State := do
  if h : i < binders.size then
    let binderViews ← matchBinder (binders.get ⟨i, h⟩)
    let s ← elabFunBinderViews binderViews 0 s
    elabFunBindersAux binders (i+1) s
  else
    pure s

end FunBinders

def elabFunBinders {α} (binders : Array Syntax) (expectedType? : Option Expr) (x : Array Expr → Option Expr → TermElabM α) : TermElabM α
  if binders.isEmpty then
    x #[] expectedType?
  else do
    let lctx ← getLCtx
    let localInsts ← getLocalInstances
    let s ← FunBinders.elabFunBindersAux binders 0 { lctx := lctx, localInsts := localInsts, expectedType? := expectedType? }
    resettingSynthInstanceCacheWhen (s.localInsts.size > localInsts.size) <| withLCtx s.lctx s.localInsts <|
      x s.fvars s.expectedType?

/- Helper function for `expandEqnsIntoMatch` -/
private def getMatchAltsNumPatterns (matchAlts : Syntax) : Nat :=
  let alt0 := matchAlts[0][0]
  let pats := alt0[1].getSepArgs
  pats.size

def expandWhereDecls (whereDecls : Syntax) (body : Syntax) : MacroM Syntax :=
  match whereDecls with
  | `(whereDecls|where $[$decls:letRecDecl $[;]?]*) => `(let rec $decls:letRecDecl,*; $body)
  | _ => Macro.throwUnsupported

def expandWhereDeclsOpt (whereDeclsOpt : Syntax) (body : Syntax) : MacroM Syntax :=
  if whereDeclsOpt.isNone then
    body
```

```
  else
    expandWhereDecls whereDeclsOpt[0] body

/- Helper function for `expandMatchAltsIntoMatch` -/
private def expandMatchAltsIntoMatchAux (matchAlts : Syntax) (matchTactic : Bool) : Nat → Array Syntax → MacroM Syntax
  | 0,   discrs => do
    if matchTactic then
      `(tactic|match $[$discrs:term],* with $matchAlts:matchAlts)
    else
      `(match $[$discrs:term],* with $matchAlts:matchAlts)
  | n+1, discrs => withFreshMacroScope do
    let x ← `(x)
    let d ← `(@$x:ident) -- See comment below
    let body ← expandMatchAltsIntoMatchAux matchAlts matchTactic n (discrs.push d)
    if matchTactic then
      `(tactic| intro $x:term; $body:tactic)
    else
      `(@fun $x => $body)

/--
  Expand `matchAlts` syntax into a full `match`-expression.
  Example
    ```
    | 0, true => alt_1
    | i, _   => alt_2
    ```
    expands into (for tactic == false)
    ```
    fun x_1 x_2 =>
    match @x_1, @x_2 with
    | 0, true => alt_1
    | i, _   => alt_2
    ```
    and (for tactic == true)
    ```
    intro x_1; intro x_2;
    match @x_1, @x_2 with
    | 0, true => alt_1
    | i, _   => alt_2
    ```

  Remark: we add `@` to make sure we don't consume implicit arguments, and to make the behavior consistent with `fun`.
  Example:
  ```
  inductive T : Type 1 :=
  | mkT : (forall {a : Type}, a -> a) -> T

```
```

```
  def makeT (f : forall {a : Type}, a -> a) : T :=
    mkT f

  def makeT' : (forall {a : Type}, a -> a) -> T
  | f => mkT f
  ```
  The two definitions should be elaborated without errors and be equivalent.
 -/
def expandMatchAltsIntoMatch (ref : Syntax) (matchAlts : Syntax) (tactic := false) : MacroM Syntax :=
  withRef ref <| expandMatchAltsIntoMatchAux matchAlts tactic (getMatchAltsNumPatterns matchAlts) #[]

def expandMatchAltsIntoMatchTactic (ref : Syntax) (matchAlts : Syntax) : MacroM Syntax :=
  withRef ref <| expandMatchAltsIntoMatchAux matchAlts true (getMatchAltsNumPatterns matchAlts) #[]

/--
  Similar to `expandMatchAltsIntoMatch`, but supports an optional `where` clause.

  Expand `matchAltsWhereDecls` into `let rec` + `match`-expression.
  Example
    ```
    | 0, true => ... f 0 ...
    | i, _    => ... f i + g i ...
    where
      f x := g x + 1

      g : Nat → Nat
        | 0   => 1
        | x+1 => f x
    ```
    expands into
    ```
    fux x_1 x_2 =>
      let rec
        f x := g x + 1,
        g : Nat → Nat
          | 0   => 1
          | x+1 => f x
      match x_1, x_2 with
      | 0, true => ... f 0 ...
      | i, _    => ... f i + g i ...
    ```
-/
def expandMatchAltsWhereDecls (matchAltsWhereDecls : Syntax) : MacroM Syntax :=
  let matchAlts   := matchAltsWhereDecls[0]
  let whereDeclsOpt := matchAltsWhereDecls[1]
  let rec loop (i : Nat) (discrs : Array Syntax) : MacroM Syntax :=
    match i with
```
```

```
    | 0   => do
      let matchStx ← `(match $[$discrs:term],* with $matchAlts:matchAlts)
      if whereDeclsOpt.isNone then
        return matchStx
      else
        expandWhereDeclsOpt whereDeclsOpt matchStx
    | n+1 => withFreshMacroScope do
      let x ← `(x)
      let d ← `(@$x:ident) -- See comment at `expandMatchAltsIntoMatch`
      let body ← loop n (discrs.push d)
      `(@fun $x => $body)
  loop (getMatchAltsNumPatterns matchAlts) #[]

@[builtinTermElab «fun»] partial def elabFun : TermElab :=
  fun stx expectedType? => loop stx expectedType?
where
  loop (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr :=
    match stx with
    | `(fun $binders* => $body) => do
      let (binders, body, expandedPattern) ← expandFunBinders binders body
      if expandedPattern then
        let newStx ← `(fun $binders* => $body)
        loop newStx expectedType?
      else
        elabFunBinders binders expectedType? fun xs expectedType? => do
          /- We ensure the expectedType here since it will force coercions to be applied if needed.
             If we just use `elabTerm`, then we will need to a coercion `Coe (α → β) (α → δ)` whenever there is a coercion `Coe β δ`,
             and another instance for the dependent version. -/
          let e ← elabTermEnsuringType body expectedType?
          mkLambdaFVars xs e
    | `(fun $m:matchAlts) => do
      let stxNew ← liftMacroM $ expandMatchAltsIntoMatch stx m
      withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?
    | _ => throwUnsupportedSyntax

/- If `useLetExpr` is true, then a kernel let-expression `let x : type := val; body` is created.
   Otherwise, we create a term of the form `(fun (x : type) => body) val`

   The default elaboration order is `binders`, `typeStx`, `valStx`, and `body`.
   If `elabBodyFirst == true`, then we use the order `binders`, `typeStx`, `body`, and `valStx`. -/
def elabLetDeclAux (id : Syntax) (binders : Array Syntax) (typeStx : Syntax) (valStx : Syntax) (body : Syntax)
    (expectedType? : Option Expr) (useLetExpr : Bool) (elabBodyFirst : Bool) : TermElabM Expr := do
  let (type, val, arity) ← elabBinders binders fun xs => do
    let type ← elabType typeStx
    registerCustomErrorIfMVar type typeStx "failed to infer 'let' declaration type"
    if elabBodyFirst then
      let type ← mkForallFVars xs type
```

```
      let val  ← mkFreshExprMVar type
      pure (type, val, xs.size)
    else
      let val  ← elabTermEnsuringType valStx type
      let type ← mkForallFVars xs type
      let val  ← mkLambdaFVars xs val
      pure (type, val, xs.size)
  trace[Elab.let.decl] "{id.getId} : {type} := {val}"
  let result ←
    if useLetExpr then
      withLetDecl id.getId type val fun x => do
        addLocalVarInfo id x
        let body ← elabTerm body expectedType?
        let body ← instantiateMVars body
        mkLetFVars #[x] body
    else
      let f ← withLocalDecl id.getId BinderInfo.default type fun x => do
        addLocalVarInfo id x
        let body ← elabTerm body expectedType?
        let body ← instantiateMVars body
        mkLambdaFVars #[x] body
      pure <| mkApp f val
  if elabBodyFirst then
    forallBoundedTelescope type arity fun xs type => do
      let valResult ← elabTermEnsuringType valStx type
      let valResult ← mkLambdaFVars xs valResult
      unless (← isDefEq val valResult) do
        throwError "unexpected error when elaborating 'let'"
  pure result

structure LetIdDeclView where
  id      : Syntax
  binders : Array Syntax
  type    : Syntax
  value   : Syntax

def mkLetIdDeclView (letIdDecl : Syntax) : LetIdDeclView :=
  -- `letIdDecl` is of the form `ident >> many bracketedBinder >> optType >> " := " >> termParser
  let id      := letIdDecl[0]
  let binders := letIdDecl[1].getArgs
  let optType := letIdDecl[2]
  let type    := expandOptType letIdDecl optType
  let value   := letIdDecl[4]
  { id := id, binders := binders, type := type, value := value }

def expandLetEqnsDecl (letDecl : Syntax) : MacroM Syntax := do
  let ref       := letDecl
```

```
  let matchAlts := letDecl[3]
  let val ← expandMatchAltsIntoMatch ref matchAlts
  return Syntax.node `Lean.Parser.Term.letIdDecl #[letDecl[0], letDecl[1], letDecl[2], mkAtomFrom ref " := ", val]

def elabLetDeclCore (stx : Syntax) (expectedType? : Option Expr) (useLetExpr : Bool) (elabBodyFirst : Bool) : TermElabM Expr := do
  let ref     := stx
  let letDecl := stx[1][0]
  let body    := stx[3]
  if letDecl.getKind == `Lean.Parser.Term.letIdDecl then
    let { id := id, binders := binders, type := type, value := val } := mkLetIdDeclView letDecl
    elabLetDeclAux id binders type val body expectedType? useLetExpr elabBodyFirst
  else if letDecl.getKind == `Lean.Parser.Term.letPatDecl then
    -- node `Lean.Parser.Term.letPatDecl  $ try (termParser >> pushNone >> optType >> " := ") >> termParser
    let pat     := letDecl[0]
    let optType := letDecl[2]
    let type    := expandOptType stx optType
    let val     := letDecl[4]
    let stxNew ← `(let x : $type := $val; match x with | $pat => $body)
    let stxNew  := match useLetExpr, elabBodyFirst with
      | true,  false => stxNew
      | true,  true  => stxNew.setKind `Lean.Parser.Term.«let_delayed»
      | false, true  => stxNew.setKind `Lean.Parser.Term.«let_fun»
      | false, false => unreachable!
    withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
  else if letDecl.getKind == `Lean.Parser.Term.letEqnsDecl then
    let letDeclIdNew ← liftMacroM <| expandLetEqnsDecl letDecl
    let declNew := stx[1].setArg 0 letDeclIdNew
    let stxNew  := stx.setArg 1 declNew
    withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
  else
    throwUnsupportedSyntax

@[builtinTermElab «let»] def elabLetDecl : TermElab :=
  fun stx expectedType? => elabLetDeclCore stx expectedType? true false

@[builtinTermElab «let_fun»] def elabLetFunDecl : TermElab :=
  fun stx expectedType? => elabLetDeclCore stx expectedType? false false

@[builtinTermElab «let_delayed»] def elabLetDelayedDecl : TermElab :=
  fun stx expectedType? => elabLetDeclCore stx expectedType? true true

builtin_initialize registerTraceClass `Elab.let

end Lean.Elab.Term
::::::::::::::
Elab/BuiltinNotation.lean
::::::::::::::
```

```
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Init.Data.ToString
import Lean.Compiler.BorrowedAnnotation
import Lean.Meta.KAbstract
import Lean.Meta.Transform
import Lean.Elab.Term
import Lean.Elab.SyntheticMVars

namespace Lean.Elab.Term
open Meta

@[builtinTermElab anonymousCtor] def elabAnonymousCtor : TermElab := fun stx expectedType? =>
  match stx with
  | `(⟨$args,*⟩) => do
    tryPostponeIfNoneOrMVar expectedType?
    match expectedType? with
    | some expectedType =>
      let expectedType ← whnf expectedType
      matchConstInduct expectedType.getAppFn
        (fun _ => throwError "invalid constructor ⟨...⟩, expected type must be an inductive type {indentExpr expectedType}")
        (fun ival us => do
          match ival.ctors with
          | [ctor] =>
            let cinfo ← getConstInfoCtor ctor
            let numExplicitFields ← forallTelescopeReducing cinfo.type fun xs _ => do
              let mut n := 0
              for i in [cinfo.numParams:xs.size] do
                if (← getFVarLocalDecl xs[i]).binderInfo.isExplicit then
                  n := n + 1
              return n
            let args := args.getElems
            if args.size < numExplicitFields then
              throwError "invalid constructor ⟨...⟩, insufficient number of arguments, constructs '{ctor}' has #{numExplicitFields} expl
            let newStx ←
              if args.size == numExplicitFields then
                `($(mkCIdentFrom stx ctor) $(args)*)
              else if numExplicitFields == 0 then
                throwError "invalid constructor ⟨...⟩, insufficient number of arguments, constructs '{ctor}' does not have explicit fiel
              else
                let extra := args[numExplicitFields-1:args.size]
                let newLast ← `(⟨$[$extra],*⟩)
                let newArgs := args[0:numExplicitFields-1].toArray.push newLast
                `($(mkCIdentFrom stx ctor) $(newArgs)*)
```

```
            withMacroExpansion stx newStx $ elabTerm newStx expectedType?
          | _ => throwError "invalid constructor ⟨...⟩, expected type must be an inductive type with only one constructor {indentExpr ex
    | none => throwError "invalid constructor ⟨...⟩, expected type must be known"
  | _ => throwUnsupportedSyntax

@[builtinTermElab borrowed] def elabBorrowed : TermElab := fun stx expectedType? =>
  match stx with
  | `(@& $e) => return markBorrowed (← elabTerm e expectedType?)
  | _ => throwUnsupportedSyntax

@[builtinMacro Lean.Parser.Term.show] def expandShow : Macro := fun stx =>
  match stx with
  | `(show $type from $val)         => let thisId := mkIdentFrom stx `this; `(let_fun $thisId : $type := $val; $thisId)
  | `(show $type by $tac:tacticSeq) => `(show $type from by $tac:tacticSeq)
  | _                               => Macro.throwUnsupported

@[builtinMacro Lean.Parser.Term.have] def expandHave : Macro := fun stx =>
  let mkId (x? : Option Syntax) : Syntax :=
    x?.getD <| mkIdentFrom stx `this
  match stx with
  | `(have $[$x :]? $type from $val $[;]? $body)              => let x := mkId x; `(let_fun $x : $type := $val; $body)
  | `(have $[$x :]? $type := $val $[;]? $body)                => let x := mkId x; `(let_fun $x : $type := $val; $body)
  | `(have $[$x :]? $type by $tac:tacticSeq $[;]? $body)      => `(have $[$x :]? $type from by $tac:tacticSeq; $body)
  | _                                                         => Macro.throwUnsupported

@[builtinMacro Lean.Parser.Term.suffices] def expandSuffices : Macro
  | `(suffices $[$x :]? $type from $val $[;]? $body)         => `(have $[$x :]? $type from $body; $val)
  | `(suffices $[$x :]? $type by $tac:tacticSeq $[;]? $body) => `(have $[$x :]? $type from $body; by $tac:tacticSeq)
  | _                                                        => Macro.throwUnsupported

private def elabParserMacroAux (prec : Syntax) (e : Syntax) : TermElabM Syntax := do
  let (some declName) ← getDeclName?
    | throwError "invalid `leading_parser` macro, it must be used in definitions"
  match extractMacroScopes declName with
  | { name := Name.str _ s _, scopes :=  scps, .. } =>
    let kind := quote declName
    let s    := quote s
    let p ← `(Lean.Parser.leadingNode $kind $prec $e)
    if scps == [] then
      -- TODO simplify the following quotation as soon as we have coercions
      `(OrElse.orElse (Lean.Parser.mkAntiquot $s (some $kind)) $p)
    else
      -- if the parser decl is hidden by hygiene, it doesn't make sense to provide an antiquotation kind
      `(OrElse.orElse (Lean.Parser.mkAntiquot $s none) $p)
  | _  => throwError "invalid `leading_parser` macro, unexpected declaration name"

@[builtinTermElab «leading_parser»] def elabLeadingParserMacro : TermElab :=
```

```
  adaptExpander fun stx => match stx with
  | `(leading_parser $e)         => elabParserMacroAux (quote Parser.maxPrec) e
  | `(leading_parser : $prec $e) => elabParserMacroAux prec e
  | _                            => throwUnsupportedSyntax

private def elabTParserMacroAux (prec lhsPrec : Syntax) (e : Syntax) : TermElabM Syntax := do
  let declName? ← getDeclName?
  match declName? with
  | some declName => let kind := quote declName; `(Lean.Parser.trailingNode $kind $prec $lhsPrec $e)
  | none          => throwError "invalid `trailing_parser` macro, it must be used in definitions"

@[builtinTermElab «trailing_parser»] def elabTrailingParserMacro : TermElab :=
  adaptExpander fun stx => match stx with
  | `(trailing_parser$[:$prec?]?$[:$lhsPrec?]? $e) =>
    elabTParserMacroAux (prec?.getD <| quote Parser.maxPrec) (lhsPrec?.getD <| quote 0) e
  | _ => throwUnsupportedSyntax

@[builtinTermElab panic] def elabPanic : TermElab := fun stx expectedType? => do
  let arg := stx[1]
  let pos ← getRefPosition
  let env ← getEnv
  let stxNew ← match (← getDeclName?) with
  | some declName => `(panicWithPosWithDecl $(quote (toString env.mainModule)) $(quote (toString declName)) $(quote pos.line) $(quote po
  | none => `(panicWithPos $(quote (toString env.mainModule)) $(quote pos.line) $(quote pos.column) $arg)
  withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?

@[builtinMacro Lean.Parser.Term.unreachable]  def expandUnreachable : Macro := fun stx =>
  `(panic! "unreachable code has been reached")

@[builtinMacro Lean.Parser.Term.assert]  def expandAssert : Macro := fun stx =>
  -- TODO: support for disabling runtime assertions
  let cond := stx[1]
  let body := stx[3]
  match cond.reprint with
  | some code => `(if $cond then $body else panic! ("assertion violation: " ++ $(quote code)))
  | none => `(if $cond then $body else panic! ("assertion violation"))

@[builtinMacro Lean.Parser.Term.dbgTrace]  def expandDbgTrace : Macro := fun stx =>
  let arg  := stx[1]
  let body := stx[3]
  if arg.getKind == interpolatedStrKind then
    `(dbgTrace (s! $arg) fun _ => $body)
  else
    `(dbgTrace (toString $arg) fun _ => $body)

@[builtinTermElab «sorry»] def elabSorry : TermElab := fun stx expectedType? => do
  logWarning "declaration uses 'sorry'"
```

```
  let stxNew ← `(sorryAx _ false)
  withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?

@[builtinTermElab emptyC] def expandEmptyC : TermElab := fun stx expectedType? => do
  let stxNew ← `(EmptyCollection.emptyCollection)
  withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?

/-- Return syntax `Prod.mk elems[0] (Prod.mk elems[1] ... (Prod.mk elems[elems.size - 2] elems[elems.size - 1])))` -/
partial def mkPairs (elems : Array Syntax) : MacroM Syntax :=
  let rec loop (i : Nat) (acc : Syntax) := do
    if i > 0 then
      let i    := i - 1
      let elem := elems[i]
      let acc ← `(Prod.mk $elem $acc)
      loop i acc
    else
      pure acc
  loop (elems.size - 1) elems.back

private partial def hasCDot : Syntax → Bool
  | Syntax.node k args =>
    if k == `Lean.Parser.Term.paren then false
    else if k == `Lean.Parser.Term.cdot then true
    else args.any hasCDot
  | _ => false

/--
  Auxiliary function for expandind the `·` notation.
  The extra state `Array Syntax` contains the new binder names.
  If `stx` is a `·`, we create a fresh identifier, store in the
  extra state, and return it. Otherwise, we just return `stx`. -/
private partial def expandCDot : Syntax → StateT (Array Syntax) MacroM Syntax
  | stx@(Syntax.node k args) =>
    if k == `Lean.Parser.Term.paren then pure stx
    else if k == `Lean.Parser.Term.cdot then withFreshMacroScope do
      let id ← `(a)
      modify fun s => s.push id;
      pure id
    else do
      let args ← args.mapM expandCDot
      pure $ Syntax.node k args
  | stx => pure stx

/--
  Return `some` if succeeded expanding `·` notation occurring in
  the given syntax. Otherwise, return `none`.
  Examples:
```

```
  - `· + 1` => `fun _a_1 => _a_1 + 1`
  - `f · · b` => `fun _a_1 _a_2 => f _a_1 _a_2 b` -/
def expandCDot? (stx : Syntax) : MacroM (Option Syntax) := do
  if hasCDot stx then
    let (newStx, binders) ← (expandCDot stx).run #[];
    `(fun $binders* => $newStx)
  else
    pure none

/--
  Try to expand `·` notation, and if successful elaborate result.
  This method is used to elaborate the Lean parentheses notation.
  Recall that in Lean the `·` notation must be surrounded by parentheses.
  We may change this is the future, but right now, here are valid examples
  - `(· + 1)`
  - `(f ⟨·, 1⟩ ·)`
  - `(· + ·)`
  - `(f · a b)` -/
private def elabCDot (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  match (← liftMacroM <| expandCDot? stx) with
  | some stx' => withMacroExpansion stx stx' (elabTerm stx' expectedType?)
  | none      => elabTerm stx expectedType?

/--
  Helper method for elaborating terms such as `(.+.)` where a constant name is expected.
  This method is usually used to implement tactics that function names as arguments (e.g., `simp`).
-/
def elabCDotFunctionAlias? (stx : Syntax) : TermElabM (Option Expr) := do
  let some stx ← liftMacroM <| expandCDotArg? stx | pure none
  let stx ← liftMacroM <| expandMacros stx
  match stx with
  | `(fun $binders* => $f:ident $args*) =>
    if binders == args then
      try Term.resolveId? f catch _ => return none
    else
      return none
  | _ => return none
where
  expandCDotArg? (stx : Syntax) : MacroM (Option Syntax) :=
    match stx with
    | `(($e)) => Term.expandCDot? e
    | _ => Term.expandCDot? stx


@[builtinTermElab paren] def elabParen : TermElab := fun stx expectedType? => do
  match stx with
  | `(())           => return Lean.mkConst `Unit.unit
```

```
  | `(($e : $type)) =>
    let type ← withSynthesize (mayPostpone := true) $ elabType type
    let e ← elabCDot e type
    ensureHasType type e
  | `(($e))         => elabCDot e expectedType?
  | `(($e, $es,*))  =>
    let pairs ← liftMacroM <| mkPairs (#[e] ++ es)
    withMacroExpansion stx pairs (elabCDot pairs expectedType?)
  | _ => throwError "unexpected parentheses notation"

@[builtinTermElab subst] def elabSubst : TermElab := fun stx expectedType? => do
  let expectedType ← tryPostponeIfHasMVars expectedType? "invalid `▸` notation"
  match stx with
  | `($heq ▸ $h) => do
     let mut heq ← elabTerm heq none
     let heqType ← inferType heq
     let heqType ← instantiateMVars heqType
     match (← Meta.matchEq? heqType) with
     | none => throwError "invalid `▸` notation, argument{indentExpr heq}\nhas type{indentExpr heqType}\nequality expected"
     | some (α, lhs, rhs) =>
       let mut lhs := lhs
       let mut rhs := rhs
       let mkMotive (typeWithLooseBVar : Expr) :=
         withLocalDeclD (← mkFreshUserName `x) α fun x => do
           mkLambdaFVars #[x] $ typeWithLooseBVar.instantiate1 x
       let mut expectedAbst ← kabstract expectedType rhs
       unless expectedAbst.hasLooseBVars do
         expectedAbst ← kabstract expectedType lhs
         unless expectedAbst.hasLooseBVars do
           throwError "invalid `▸` notation, expected type{indentExpr expectedType}\ndoes contain equation left-hand-side nor right-hand
         heq ← mkEqSymm heq
         (lhs, rhs) := (rhs, lhs)
       let hExpectedType := expectedAbst.instantiate1 lhs
       let h ← withRef h do
         let h ← elabTerm h hExpectedType
         try
           ensureHasType hExpectedType h
         catch ex =>
           -- if `rhs` occurs in `hType`, we try to apply `heq` to `h` too
           let hType ← inferType h
           let hTypeAbst ← kabstract hType rhs
           unless hTypeAbst.hasLooseBVars do
             throw ex
           let hTypeNew := hTypeAbst.instantiate1 lhs
           unless (← isDefEq hExpectedType hTypeNew) do
             throw ex
           mkEqNDRec (← mkMotive hTypeAbst) h (← mkEqSymm heq)
```

```
      mkEqNDRec (← mkMotive expectedAbst) h heq
  | _ => throwUnsupportedSyntax

@[builtinTermElab stateRefT] def elabStateRefT : TermElab := fun stx _ => do
  let σ ← elabType stx[1]
  let mut m := stx[2]
  if m.getKind == `Lean.Parser.Term.macroDollarArg then
    m := m[1]
  let m ← elabTerm m (← mkArrow (mkSort levelOne) (mkSort levelOne))
  let ω ← mkFreshExprMVar (mkSort levelOne)
  let stWorld ← mkAppM `STWorld #[ω, m]
  discard <| mkInstMVar stWorld
  mkAppM `StateRefT' #[ω, σ, m]

@[builtinTermElab noindex] def elabNoindex : TermElab := fun stx expectedType? => do
  let e ← elabTerm stx[1] expectedType?
  return DiscrTree.mkNoindexAnnotation e

end Lean.Elab.Term
::::::::::::::
Elab/Command.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Command
import Lean.ResolveName
import Lean.Meta.Reduce
import Lean.Elab.Log
import Lean.Elab.Term
import Lean.Elab.Binders
import Lean.Elab.SyntheticMVars
import Lean.Elab.DeclModifiers
import Lean.Elab.InfoTree
import Lean.Elab.Open
import Lean.Elab.SetOption

namespace Lean.Elab.Command

structure Scope where
  header        : String
  opts          : Options := {}
  currNamespace : Name := Name.anonymous
  openDecls     : List OpenDecl := []
  levelNames    : List Name := []
```

```
  /-- section variables as `bracketedBinder`s -/
  varDecls      : Array Syntax := #[]
  /-- Globally unique internal identifiers for the `varDecls` -/
  varUIds       : Array Name := #[]
  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 := {}
  infoState      : InfoState := {}
  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

def mkState (env : Environment) (messages : MessageLog := {}) (opts : Options := {}) : State := {
  env         := env
  messages    := messages
  scopes      := [{ header := "", opts := opts }]
  maxRecDepth := maxRecDepth.get opts
}

/- Linters should be loadable as plugins, so store in a global IO ref instead of an attribute managed by the
    environment (which only contains `import`ed objects). -/
builtin_initialize lintersRef : IO.Ref (Array Linter) ← IO.mkRef #[]

def addLinter (l : Linter) : IO Unit := do
  let ls ← lintersRef.get
  lintersRef.set (ls.push l)

instance : MonadInfoTree CommandElabM where
```

```
  getInfoState      := return (← get).infoState
  modifyInfoState f := modify fun s => { s with infoState := f s.infoState }

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

instance : MonadOptions CommandElabM where
  getOptions := do pure (← get).scopes.head!.opts

protected def getRef : CommandElabM Syntax :=
  return (← read).ref

instance : AddMessageContext CommandElabM where
  addMessageContext := addMessageContextPartial

instance : MonadRef CommandElabM where
  getRef := Command.getRef
  withRef ref x := withReader (fun ctx => { ctx with ref := ref }) x

instance : AddErrorMessageContext CommandElabM where
  add ref msg := do
    let ctx ← read
    let ref := getBetterRef ref ctx.macroStack
    let msg ← addMessageContext msg
    let msg ← addMacroStack msg ctx.macroStack
    return (ref, msg)

def mkMessageAux (ctx : Context) (ref : Syntax) (msgData : MessageData) (severity : MessageSeverity) : Message :=
  mkMessageCore ctx.fileName ctx.fileMap msgData severity (ref.getPos?.getD ctx.cmdPos)

private def mkCoreContext (ctx : Context) (s : State) (heartbeats : Nat) : Core.Context :=
  let scope        := s.scopes.head!
  { options        := scope.opts
    currRecDepth   := ctx.currRecDepth
    maxRecDepth    := s.maxRecDepth
    ref            := ctx.ref
    currNamespace  := scope.currNamespace
    openDecls      := scope.openDecls
    initHeartbeats := heartbeats }

def liftCoreM {α} (x : CoreM α) : CommandElabM α := do
  let s ← get
  let ctx ← read
  let heartbeats ← IO.getNumHeartbeats (ε := Exception)
  let Eα := Except Exception α
  let x : CoreM Eα := try let a ← x; pure <| Except.ok a catch ex => pure <| Except.error ex
```

```
  let x : EIO Exception (Eα × Core.State) := (ReaderT.run x (mkCoreContext ctx s heartbeats)).run { env := s.env, ngen := s.ngen }
  let (ea, coreS) ← liftM x
  modify fun s => { s with env := coreS.env, ngen := coreS.ngen }
  match ea with
  | Except.ok a    => pure a
  | Except.error e => throw e

private def ioErrorToMessage (ctx : Context) (ref : Syntax) (err : IO.Error) : Message :=
  let ref := getBetterRef ref ctx.macroStack
  mkMessageAux ctx ref (toString err) MessageSeverity.error

@[inline] def liftEIO {α} (x : EIO Exception α) : CommandElabM α := liftM x

@[inline] def liftIO {α} (x : IO α) : CommandElabM α := do
  let ctx ← read
  IO.toEIO (fun (ex : IO.Error) => Exception.error ctx.ref ex.toString) x

instance : MonadLiftT IO CommandElabM where
  monadLift := liftIO

def getScope : CommandElabM Scope := do pure (← get).scopes.head!

instance : MonadResolveName CommandElabM where
  getCurrNamespace := return (← getScope).currNamespace
  getOpenDecls     := return (← getScope).openDecls

instance : MonadLog CommandElabM where
  getRef      := getRef
  getFileMap  := return (← read).fileMap
  getFileName := return (← read).fileName
  logMessage msg := do
    let currNamespace ← getCurrNamespace
    let openDecls ← getOpenDecls
    let msg := { msg with data := MessageData.withNamingContext { currNamespace := currNamespace, openDecls := openDecls } msg.data }
    modify fun s => { s with messages := s.messages.add msg }

def runLinters (stx : Syntax) : CommandElabM Unit := do
  let linters ← lintersRef.get
  unless linters.isEmpty do
    for linter in linters do
      let savedState ← get
      try
        linter stx
      catch ex =>
        logException ex
      finally
        modify fun s => { savedState with messages := s.messages }
```

```
protected def getCurrMacroScope : CommandElabM Nat  := do pure (← read).currMacroScope
protected def getMainModule     : CommandElabM Name := do pure (← getEnv).mainModule

@[inline] protected def withFreshMacroScope {α} (x : CommandElabM α) : CommandElabM α := do
  let fresh ← modifyGet (fun st => (st.nextMacroScope, { st with nextMacroScope := st.nextMacroScope + 1 }))
  withReader (fun ctx => { ctx with currMacroScope := fresh }) x

instance : MonadQuotation CommandElabM where
  getCurrMacroScope   := Command.getCurrMacroScope
  getMainModule       := Command.getMainModule
  withFreshMacroScope := Command.withFreshMacroScope

unsafe def mkCommandElabAttributeUnsafe : IO (KeyedDeclsAttribute CommandElab) :=
  mkElabAttribute CommandElab `Lean.Elab.Command.commandElabAttribute `builtinCommandElab `commandElab `Lean.Parser.Command `Lean.Elab.C

@[implementedBy mkCommandElabAttributeUnsafe]
constant mkCommandElabAttribute : IO (KeyedDeclsAttribute CommandElab)

builtin_initialize commandElabAttribute : KeyedDeclsAttribute CommandElab ← mkCommandElabAttribute

private def elabCommandUsing (s : State) (stx : Syntax) : List CommandElab → CommandElabM Unit
  | []                => throwError "unexpected syntax{indentD stx}"
  | (elabFn::elabFns) =>
    catchInternalId unsupportedSyntaxExceptionId
      (elabFn stx)
      (fun _ => do set s; elabCommandUsing s stx elabFns)

/- Elaborate `x` with `stx` on the macro stack -/
@[inline] def withMacroExpansion {α} (beforeStx afterStx : Syntax) (x : CommandElabM α) : CommandElabM α :=
  withReader (fun ctx => { ctx with macroStack := { before := beforeStx, after := afterStx } :: ctx.macroStack }) x

instance : MonadMacroAdapter CommandElabM where
  getCurrMacroScope := getCurrMacroScope
  getNextMacroScope := return (← get).nextMacroScope
  setNextMacroScope next := modify fun s => { s with nextMacroScope := next }

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

register_builtin_option showPartialSyntaxErrors : Bool := {
  defValue := false
  descr    := "show elaboration errors from partial syntax trees (i.e. after parser recovery)"
}
```

```
@[inline] def withLogging (x : CommandElabM Unit) : CommandElabM Unit := do
  try
    x
  catch ex => match ex with
    | Exception.error _ _     => logException ex
    | Exception.internal id _ =>
      if isAbortExceptionId id then
        pure ()
      else
        let idName ← liftIO <| id.getName;
        logError m!"internal exception {idName}"

builtin_initialize registerTraceClass `Elab.command

partial def elabCommand (stx : Syntax) : CommandElabM Unit := do
  let mkInfoTree trees := do
    let ctx ← read
    let s ← get
    let scope := s.scopes.head!
    let tree := InfoTree.node (Info.ofCommandInfo { stx := stx }) trees
    let tree := InfoTree.context {
      env := s.env, fileMap := ctx.fileMap, mctx := {}, currNamespace := scope.currNamespace, openDecls := scope.openDecls, options := s
    } tree
    if checkTraceOption (← getOptions) `Elab.info then
      logTrace `Elab.info m!"{← tree.format}"
    return tree
  let initMsgs ← modifyGet fun st => (st.messages, { st with messages := {} })
  withLogging <| withRef stx <| withInfoTreeContext (mkInfoTree := mkInfoTree) <| withIncRecDepth <| withFreshMacroScope do
    runLinters stx
    match stx with
    | Syntax.node k args =>
      if k == nullKind then
        -- list of commands => elaborate in order
        -- The parser will only ever return a single command at a time, but syntax quotations can return multiple ones
        args.forM elabCommand
      else do
        trace `Elab.command fun _ => stx;
        let s ← get
        let stxNew? ← catchInternalId unsupportedSyntaxExceptionId
          (do let newStx ← adaptMacro (getMacros s.env) stx; pure (some newStx))
          (fun ex => pure none)
        match stxNew? with
        | some stxNew => withMacroExpansion stx stxNew <| elabCommand stxNew
        | _ =>
          let table := (commandElabAttribute.ext.getState s.env).table;
          let k := stx.getKind;
          match table.find? k with
```

```
        | some elabFns => elabCommandUsing s stx elabFns
        | none         => throwError "elaboration function for '{k}' has not been implemented"
    | _ => throwError "unexpected command"
  let mut msgs ← (← get).messages
  -- `stx.hasMissing` should imply `initMsgs.hasErrors`, but the latter should be cheaper to check in general
  if !showPartialSyntaxErrors.get (← getOptions) && initMsgs.hasErrors && stx.hasMissing then
    -- discard elaboration errors, except for a few important and unlikely misleading ones, on parse error
    msgs := ⟨msgs.msgs.filter fun msg =>
      msg.data.hasTag `Elab.synthPlaceholder || msg.data.hasTag `Tactic.unsolvedGoals⟩
  modify ({ · with messages := initMsgs ++ msgs })

/-- Adapt a syntax transformation to a regular, command-producing elaborator. -/
def adaptExpander (exp : Syntax → CommandElabM Syntax) : CommandElab := fun stx => do
  let stx' ← exp stx
  withMacroExpansion stx stx' <| elabCommand stx'

private def getVarDecls (s : State) : Array Syntax :=
  s.scopes.head!.varDecls

instance {α} : Inhabited (CommandElabM α) where
  default := throw arbitrary

private def mkMetaContext : Meta.Context := {
  config := { foApprox := true, ctxApprox := true, quasiPatternApprox := true }
}

def getBracketedBinderIds : Syntax → Array Name
  | `(bracketedBinder|($ids* $[: $ty?]? $(annot?)?)) => ids.map Syntax.getId
  | `(bracketedBinder|{$ids* $[: $ty?]?})            => ids.map Syntax.getId
  | `(bracketedBinder|[$id : $ty])                   => #[id.getId]
  | `(bracketedBinder|[$ty])                         => #[]
  | _                                                => #[]

private def mkTermContext (ctx : Context) (s : State) (declName? : Option Name) : Term.Context := do
  let scope      := s.scopes.head!
  let mut sectionVars := {}
  for id in scope.varDecls.concatMap getBracketedBinderIds, uid in scope.varUIds do
    sectionVars := sectionVars.insert id uid
  { macroStack     := ctx.macroStack
    fileName       := ctx.fileName
    fileMap        := ctx.fileMap
    currMacroScope := ctx.currMacroScope
    declName?      := declName?
    sectionVars    := sectionVars }

private def mkTermState (scope : Scope) (s : State) : Term.State := {
  messages          := {}
```

```
  levelNames        := scope.levelNames
  infoState.enabled := s.infoState.enabled
}

private def addTraceAsMessages (ctx : Context) (log : MessageLog) (traceState : TraceState) : MessageLog :=
  traceState.traces.foldl (init := log) fun (log : MessageLog) traceElem =>
      let ref := replaceRef traceElem.ref ctx.ref;
      let pos := ref.getPos?.getD 0;
      log.add (mkMessageCore ctx.fileName ctx.fileMap traceElem.msg MessageSeverity.information pos)

def liftTermElabM {α} (declName? : Option Name) (x : TermElabM α) : CommandElabM α := do
  let ctx ← read
  let s   ← get
  let heartbeats ← IO.getNumHeartbeats (ε := Exception)
  -- dbg_trace "heartbeats: {heartbeats}"
  let scope := s.scopes.head!
  -- We execute `x` with an empty message log. Thus, `x` cannot modify/view messages produced by previous commands.
  -- This is useful for implementing `runTermElabM` where we use `Term.resetMessageLog`
  let x : MetaM _      := (observing x).run (mkTermContext ctx s declName?) (mkTermState scope s)
  let x : CoreM _      := x.run mkMetaContext {}
  let x : EIO _ _      := x.run (mkCoreContext ctx s heartbeats) { env := s.env, ngen := s.ngen, nextMacroScope := s.nextMacroScope }
  let (((ea, termS), metaS), coreS) ← liftEIO x
  let infoTrees        := termS.infoState.trees.map fun tree =>
    let tree := tree.substitute termS.infoState.assignment
    InfoTree.context {
      env := coreS.env, fileMap := ctx.fileMap, mctx := metaS.mctx, currNamespace := scope.currNamespace, openDecls := scope.openDecls,
    } tree
  modify fun s => { s with
    env               := coreS.env
    messages          := addTraceAsMessages ctx (s.messages ++ termS.messages) coreS.traceState
    nextMacroScope    := coreS.nextMacroScope
    ngen              := coreS.ngen
    infoState.trees := s.infoState.trees.append infoTrees
  }
  match ea with
  | Except.ok a     => pure a
  | Except.error ex => throw ex

@[inline] def runTermElabM {α} (declName? : Option Name) (elabFn : Array Expr → TermElabM α) : CommandElabM α := do
  let scope ← getScope
  liftTermElabM declName? <|
    Term.withAutoBoundImplicit <|
      Term.elabBinders scope.varDecls fun xs => do
        -- We need to synthesize postponed terms because this is a checkpoint for the auto-bound implicit feature
        -- If we don't use this checkpoint here, then auto-bound implicits in the postponed terms will not be handled correctly.
        Term.synthesizeSyntheticMVarsNoPostponing
        let mut sectionFVars := {}
```

```
        for uid in scope.varUIds, x in xs do
          sectionFVars := sectionFVars.insert uid x
        withReader ({ · with sectionFVars := sectionFVars }) do
          -- We don't want to store messages produced when elaborating `(getVarDecls s)` because they have already been saved when we el
          -- So, we use `Term.resetMessageLog`.
          Term.resetMessageLog
          let xs ← Term.addAutoBoundImplicits xs
          Term.withoutAutoBoundImplicit <| elabFn xs

@[inline] def catchExceptions (x : CommandElabM Unit) : CommandElabCoreM Empty Unit := fun ctx ref =>
  EIO.catchExceptions (withLogging x ctx ref) (fun _ => pure ())

private def liftAttrM {α} (x : AttrM α) : CommandElabM α := do
  liftCoreM x

private def addScope (isNewNamespace : Bool) (header : String) (newNamespace : Name) : CommandElabM Unit := do
  modify fun s => { s with
    env    := s.env.registerNamespace newNamespace,
    scopes := { s.scopes.head! with header := header, currNamespace := newNamespace } :: s.scopes
  }
  pushScope
  if isNewNamespace then
    activateScoped newNamespace

private def addScopes (isNewNamespace : Bool) : Name → CommandElabM Unit
  | Name.anonymous => pure ()
  | Name.str p header _ => do
    addScopes isNewNamespace p
    let currNamespace ← getCurrNamespace
    addScope isNewNamespace header (if isNewNamespace then Name.mkStr currNamespace header else currNamespace)
  | _ => throwError "invalid scope"

private def addNamespace (header : Name) : CommandElabM Unit :=
  addScopes (isNewNamespace := true) header

@[builtinCommandElab «namespace»] def elabNamespace : CommandElab := fun stx =>
  match stx with
  | `(namespace $n) => addNamespace n.getId
  | _               => throwUnsupportedSyntax

@[builtinCommandElab «section»] def elabSection : CommandElab := fun stx =>
  match stx with
  | `(section $header:ident) => addScopes (isNewNamespace := false) header.getId
  | `(section)               => do let currNamespace ← getCurrNamespace; addScope (isNewNamespace := false) "" currNamespace
  | _                        => throwUnsupportedSyntax

def getScopes : CommandElabM (List Scope) := do
```

```
  pure (← get).scopes

private def checkAnonymousScope : List Scope → Bool
  | { header := "", .. } :: _   => true
  | _                           => false

private def checkEndHeader : Name → List Scope → Bool
  | Name.anonymous, _                                 => true
  | Name.str p s _, { header := h, .. } :: scopes => h == s && checkEndHeader p scopes
  | _,              _                                 => false

private def popScopes (numScopes : Nat) : CommandElabM Unit :=
  for i in [0:numScopes] do
    popScope

@[builtinCommandElab «end»] def elabEnd : CommandElab := fun stx => do
  let header? := (stx.getArg 1).getOptionalIdent?;
  let endSize := match header? with
    | none   => 1
    | some n => n.getNumParts
  let scopes ← getScopes
  if endSize < scopes.length then
    modify fun s => { s with scopes := s.scopes.drop endSize }
    popScopes endSize
  else -- we keep "root" scope
    let n := (← get).scopes.length - 1
    modify fun s => { s with scopes := s.scopes.drop n }
    popScopes n
    throwError "invalid 'end', insufficient scopes"
  match header? with
  | none        =>
    unless checkAnonymousScope scopes do
      throwError "invalid 'end', name is missing"
  | some header =>
    unless checkEndHeader header scopes do
      addCompletionInfo <| CompletionInfo.endSection stx (scopes.map fun scope => scope.header)
      throwError "invalid 'end', name mismatch"

@[inline] def withNamespace {α} (ns : Name) (elabFn : CommandElabM α) : CommandElabM α := do
  addNamespace ns
  let a ← elabFn
  modify fun s => { s with scopes := s.scopes.drop ns.getNumParts }
  pure a

@[specialize] def modifyScope (f : Scope → Scope) : CommandElabM Unit :=
  modify fun s => { s with
    scopes := match s.scopes with
```

```
    | h::t => f h :: t
    | []   => unreachable!
  }

def getLevelNames : CommandElabM (List Name) :=
  return (← getScope).levelNames

def addUnivLevel (idStx : Syntax) : CommandElabM Unit := withRef idStx do
  let id := idStx.getId
  let levelNames ← getLevelNames
  if levelNames.elem id then
    throwAlreadyDeclaredUniverseLevel id
  else
    modifyScope fun scope => { scope with levelNames := id :: scope.levelNames }

partial def elabChoiceAux (cmds : Array Syntax) (i : Nat) : CommandElabM Unit :=
  if h : i < cmds.size then
    let cmd := cmds.get ⟨i, h⟩;
    catchInternalId unsupportedSyntaxExceptionId
      (elabCommand cmd)
      (fun ex => elabChoiceAux cmds (i+1))
  else
    throwUnsupportedSyntax

@[builtinCommandElab choice] def elbChoice : CommandElab := fun stx =>
  elabChoiceAux stx.getArgs 0

@[builtinCommandElab «universe»] def elabUniverse : CommandElab := fun n => do
  addUnivLevel n[1]

@[builtinCommandElab «universes»] def elabUniverses : CommandElab := fun n => do
  n[1].forArgsM addUnivLevel

@[builtinCommandElab «init_quot»] def elabInitQuot : CommandElab := fun stx => do
  match (← getEnv).addDecl Declaration.quotDecl with
  | Except.ok env   => setEnv env
  | Except.error ex => throwError (ex.toMessageData (← getOptions))

@[builtinCommandElab «export»] def elabExport : CommandElab := fun stx => do
  -- `stx` is of the form (Command.export "export" <namespace> "(" (null <ids>*) ")")
  let id  := stx[1].getId
  let ns ← resolveNamespace id
  let currNamespace ← getCurrNamespace
  if ns == currNamespace then throwError "invalid 'export', self export"
  let env ← getEnv
  let ids := stx[3].getArgs
  let aliases ← ids.foldlM (init := []) fun (aliases : List (Name × Name)) (idStx : Syntax) => do
```

```
    let id := idStx.getId
    let declName := ns ++ id
    if env.contains declName then
      pure <| (currNamespace ++ id, declName) :: aliases
    else
      withRef idStx <| logUnknownDecl declName
      pure aliases
  modify fun s => { s with env := aliases.foldl (init := s.env) fun env p => addAlias env p.1 p.2 }

@[builtinCommandElab «open»] def elabOpen : CommandElab := fun n => do
  let openDecls ← elabOpenDecl n[1]
  modifyScope fun scope => { scope with openDecls := openDecls }

@[builtinCommandElab «variable»] def elabVariable : CommandElab
  | `(variable $binders*) => do
    -- Try to elaborate `binders` for sanity checking
    runTermElabM none fun _ => Term.withAutoBoundImplicit <|
      Term.elabBinders binders fun _ => pure ()
    let varUIds ← binders.concatMap getBracketedBinderIds |>.mapM (withFreshMacroScope ∘ MonadQuotation.addMacroScope)
    modifyScope fun scope => { scope with varDecls := scope.varDecls ++ binders, varUIds := scope.varUIds ++ varUIds }
  | _ => throwUnsupportedSyntax

open Meta

@[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

@[builtinCommandElab Lean.Parser.Command.reduce] def elabReduce : CommandElab
  | `(#reduce%$tk $term) => withoutModifyingEnv <| runTermElabM (some `_check) fun _ => do
    let e ← Term.elabTerm term none
    Term.synthesizeSyntheticMVarsNoPostponing
    let (e, _) ← Term.levelMVarToParam (← instantiateMVars e)
    -- TODO: add options or notation for setting the following parameters
    withTheReader Core.Context (fun ctx => { ctx with options := ctx.options.setBool `smartUnfolding false }) do
      let e ← withTransparency (mode := TransparencyMode.all) <| reduce e (skipProofs := false) (skipTypes := false)
      logInfoAt tk e
  | _ => throwUnsupportedSyntax

def hasNoErrorMessages : CommandElabM Bool := do
  return !(← get).messages.hasErrors
```

```
def failIfSucceeds (x : CommandElabM Unit) : CommandElabM Unit := do
  let resetMessages : CommandElabM MessageLog := do
    let s ← get
    let messages := s.messages;
    modify fun s => { s with messages := {} };
    pure messages
  let restoreMessages (prevMessages : MessageLog) : CommandElabM Unit := do
    modify fun s => { s with messages := prevMessages ++ s.messages.errorsToWarnings }
  let prevMessages ← resetMessages
  let succeeded ←
    try
      x
      hasNoErrorMessages
    catch
      | ex@(Exception.error _ _) => do logException ex; pure false
      | Exception.internal id _  => do logError (← id.getName); pure false
    finally
      restoreMessages prevMessages
  if succeeded then
    throwError "unexpected success"

@[builtinCommandElab «check_failure»] def elabCheckFailure : CommandElab
  | `(#check_failure $term) => do
    failIfSucceeds <| elabCheck (← `(#check $term))
  | _ => throwUnsupportedSyntax

unsafe def elabEvalUnsafe : CommandElab
  | `(#eval%$tk $term) => do
    let n := `_eval
    let ctx ← read
    let addAndCompile (value : Expr) : TermElabM Unit := do
      let type ← inferType value
      let decl := Declaration.defnDecl {
        name        := n
        levelParams := []
        type        := type
        value       := value
        hints       := ReducibilityHints.opaque
        safety      := DefinitionSafety.unsafe
      }
      Term.ensureNoUnassignedMVars decl
      addAndCompile decl
    let elabMetaEval : CommandElabM Unit := runTermElabM (some n) fun _ => do
      let e ← Term.elabTerm term none
      Term.synthesizeSyntheticMVarsNoPostponing
      let e ← withLocalDeclD `env (mkConst ``Lean.Environment) fun env =>
```

```
     withLocalDeclD `opts (mkConst ``Lean.Options) fun opts => do
      let e ← mkAppM ``Lean.runMetaEval #[env, opts, e];
      mkLambdaFVars #[env, opts] e
   let env ← getEnv
   let opts ← getOptions
   let act ← try addAndCompile e; evalConst (Environment → Options → IO (String × Except IO.Error Environment)) n finally setEnv env
   let (out, res) ← act env opts -- we execute `act` using the environment
   logInfoAt tk out
   match res with
   | Except.error e => throwError e.toString
   | Except.ok env  => do setEnv env; pure ()
  let elabEval : CommandElabM Unit := runTermElabM (some n) fun _ => do
   -- fall back to non-meta eval if MetaEval hasn't been defined yet
   -- modify e to `runEval e`
   let e ← Term.elabTerm term none
   let e := mkSimpleThunk e
   Term.synthesizeSyntheticMVarsNoPostponing
   let e ← mkAppM ``Lean.runEval #[e]
   let env ← getEnv
   let act ← try addAndCompile e; evalConst (IO (String × Except IO.Error Unit)) n finally setEnv env
   let (out, res) ← liftM (m := IO) act
   logInfoAt tk out
   match res with
   | Except.error e => throwError e.toString
   | Except.ok _    => pure ()
  if (← getEnv).contains ``Lean.MetaEval then do
   elabMetaEval
  else
   elabEval
 | _ => throwUnsupportedSyntax

@[builtinCommandElab «eval», implementedBy elabEvalUnsafe]
constant elabEval : CommandElab

@[builtinCommandElab «synth»] def elabSynth : CommandElab := fun stx => do
 let term := stx[1]
 withoutModifyingEnv <| runTermElabM `_synth_cmd fun _ => do
  let inst ← Term.elabTerm term none
  Term.synthesizeSyntheticMVarsNoPostponing
  let inst ← instantiateMVars inst
  let val  ← synthInstance inst
  logInfo val
  pure ()

@[builtinCommandElab «set_option»] def elabSetOption : CommandElab := fun stx => do
 let options ← Elab.elabSetOption stx[1] stx[2]
 modify fun s => { s with maxRecDepth := maxRecDepth.get options }
```

```
  modifyScope fun scope => { scope with opts := options }

@[builtinMacro Lean.Parser.Command.«in»] def expandInCmd : Macro := fun stx => do
  let cmd₁ := stx[0]
  let cmd₂ := stx[2]
  `(section $cmd₁:command $cmd₂:command end)

def expandDeclId (declId : Syntax) (modifiers : Modifiers) : CommandElabM ExpandDeclIdResult := do
  let currNamespace ← getCurrNamespace
  let currLevelNames ← getLevelNames
  Lean.Elab.expandDeclId currNamespace currLevelNames declId modifiers

end Elab.Command

export Elab.Command (Linter addLinter)

end Lean
::::::::::::::
Elab/Declaration.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Util.CollectLevelParams
import Lean.Elab.DeclUtil
import Lean.Elab.DefView
import Lean.Elab.Inductive
import Lean.Elab.Structure
import Lean.Elab.MutualDef
import Lean.Elab.DeclarationRange
namespace Lean.Elab.Command

open Meta

/- Auxiliary function for `expandDeclNamespace?` -/
def expandDeclIdNamespace? (declId : Syntax) : Option (Name × Syntax) :=
  let (id, optUnivDeclStx) := expandDeclIdCore declId
  let scpView := extractMacroScopes id
  match scpView.name with
  | Name.str Name.anonymous s _ => none
  | Name.str pre s _            =>
    let nameNew := { scpView with name := Name.mkSimple s }.review
    if declId.isIdent then
      some (pre, mkIdentFrom declId nameNew)
    else
```

```
      some (pre, declId.setArg 0 (mkIdentFrom declId nameNew))
  | _ => none

/- given declarations such as `@[...] def Foo.Bla.f ...` return `some (Foo.Bla, @[...] def f ...)` -/
def expandDeclNamespace? (stx : Syntax) : Option (Name × Syntax) :=
  if !stx.isOfKind `Lean.Parser.Command.declaration then none
  else
    let decl := stx[1]
    let k := decl.getKind
    if k == `Lean.Parser.Command.abbrev ||
       k == `Lean.Parser.Command.def ||
       k == `Lean.Parser.Command.theorem ||
       k == `Lean.Parser.Command.constant ||
       k == `Lean.Parser.Command.axiom ||
       k == `Lean.Parser.Command.inductive ||
       k == `Lean.Parser.Command.classInductive ||
       k == `Lean.Parser.Command.structure then
      match expandDeclIdNamespace? decl[1] with
      | some (ns, declId) => some (ns, stx.setArg 1 (decl.setArg 1 declId))
      | none              => none
    else if k == `Lean.Parser.Command.instance then
      let optDeclId := decl[3]
      if optDeclId.isNone then none
      else match expandDeclIdNamespace? optDeclId[0] with
        | some (ns, declId) => some (ns, stx.setArg 1 (decl.setArg 3 (optDeclId.setArg 0 declId)))
        | none              => none
    else
      none

def elabAxiom (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
  -- leading_parser "axiom " >> declId >> declSig
  let declId             := stx[1]
  let (binders, typeStx) := expandDeclSig stx[2]
  let scopeLevelNames ← getLevelNames
  let ⟨name, declName, allUserLevelNames⟩ ← expandDeclId declId modifiers
  addDeclarationRanges declName stx
  runTermElabM declName fun vars => Term.withLevelNames allUserLevelNames $ Term.elabBinders binders.getArgs fun xs => do
    Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.beforeElaboration
    let type ← Term.elabType typeStx
    Term.synthesizeSyntheticMVarsNoPostponing
    let type ← instantiateMVars type
    let type ← mkForallFVars xs type
    let type ← mkForallFVars vars type (usedOnly := true)
    let (type, _) ← Term.levelMVarToParam type
    let usedParams  := collectLevelParams {} type |>.params
    match sortDeclLevelParams scopeLevelNames allUserLevelNames usedParams with
    | Except.error msg      => throwErrorAt stx msg
```

```
    | Except.ok levelParams =>
      let decl := Declaration.axiomDecl {
        name        := declName,
        levelParams := levelParams,
        type        := type,
        isUnsafe    := modifiers.isUnsafe
      }
      Term.ensureNoUnassignedMVars decl
      addDecl decl
      Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.afterTypeChecking
      if isExtern (← getEnv) declName then
        compileDecl decl
      Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.afterCompilation

/-
leading_parser "inductive " >> declId >> optDeclSig >> optional ":=" >> many ctor
leading_parser atomic (group ("class " >> "inductive ")) >> declId >> optDeclSig >> optional ":=" >> many ctor >> optDeriving
-/
private def inductiveSyntaxToView (modifiers : Modifiers) (decl : Syntax) : CommandElabM InductiveView := do
  checkValidInductiveModifier modifiers
  let (binders, type?) := expandOptDeclSig decl[2]
  let declId           := decl[1]
  let ⟨name, declName, levelNames⟩ ← expandDeclId declId modifiers
  addDeclarationRanges declName decl
  let ctors      ← decl[4].getArgs.mapM fun ctor => withRef ctor do
    -- def ctor := leading_parser " | " >> declModifiers >> ident >> optional inferMod >> optDeclSig
    let ctorModifiers ← elabModifiers ctor[1]
    if ctorModifiers.isPrivate && modifiers.isPrivate then
      throwError "invalid 'private' constructor in a 'private' inductive datatype"
    if ctorModifiers.isProtected && modifiers.isPrivate then
      throwError "invalid 'protected' constructor in a 'private' inductive datatype"
    checkValidCtorModifier ctorModifiers
    let ctorName := ctor.getIdAt 2
    let ctorName := declName ++ ctorName
    let ctorName ← withRef ctor[2] $ applyVisibility ctorModifiers.visibility ctorName
    let inferMod := !ctor[3].isNone
    let (binders, type?) := expandOptDeclSig ctor[4]
    addDocString' ctorName ctorModifiers.docString?
    addAuxDeclarationRanges ctorName ctor ctor[2]
    pure { ref := ctor, modifiers := ctorModifiers, declName := ctorName, inferMod := inferMod, binders := binders, type? := type? : Cto
  let classes ← getOptDerivingClasses decl[5]
  pure {
    ref             := decl
    modifiers       := modifiers
    shortDeclName   := name
    declName        := declName
    levelNames      := levelNames
```

```
    binders         := binders
    type?           := type?
    ctors           := ctors
    derivingClasses := classes
  }

private def classInductiveSyntaxToView (modifiers : Modifiers) (decl : Syntax) : CommandElabM InductiveView :=
  inductiveSyntaxToView modifiers decl

def elabInductive (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
  let v ← inductiveSyntaxToView modifiers stx
  elabInductiveViews #[v]

def elabClassInductive (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
  let modifiers := modifiers.addAttribute { name := `class }
  let v ← classInductiveSyntaxToView modifiers stx
  elabInductiveViews #[v]

@[builtinCommandElab declaration]
def elabDeclaration : CommandElab := fun stx =>
  match expandDeclNamespace? stx with
  | some (ns, newStx) => do
    let ns := mkIdentFrom stx ns
    let newStx ← `(namespace $ns:ident $newStx end $ns:ident)
    withMacroExpansion stx newStx $ elabCommand newStx
  | none => do
    let modifiers ← elabModifiers stx[0]
    let decl     := stx[1]
    let declKind := decl.getKind
    if declKind == `Lean.Parser.Command.«axiom» then
      elabAxiom modifiers decl
    else if declKind == `Lean.Parser.Command.«inductive» then
      elabInductive modifiers decl
    else if declKind == `Lean.Parser.Command.classInductive then
      elabClassInductive modifiers decl
    else if declKind == `Lean.Parser.Command.«structure» then
      elabStructure modifiers decl
    else if isDefLike decl then
      elabMutualDef #[stx]
    else
      throwError "unexpected declaration"

/- Return true if all elements of the mutual-block are inductive declarations. -/
private def isMutualInductive (stx : Syntax) : Bool :=
  stx[1].getArgs.all fun elem =>
    let decl     := elem[1]
    let declKind := decl.getKind
```

```
    declKind == `Lean.Parser.Command.inductive

private def elabMutualInductive (elems : Array Syntax) : CommandElabM Unit := do
  let views ← elems.mapM fun stx => do
     let modifiers ← elabModifiers stx[0]
     inductiveSyntaxToView modifiers stx[1]
  elabInductiveViews views

/- Return true if all elements of the mutual-block are definitions/theorems/abbrevs. -/
private def isMutualDef (stx : Syntax) : Bool :=
  stx[1].getArgs.all fun elem =>
    let decl := elem[1]
    isDefLike decl

private def isMutualPreambleCommand (stx : Syntax) : Bool :=
  let k := stx.getKind
  k == `Lean.Parser.Command.variable ||
  k == `Lean.Parser.Command.variables ||
  k == `Lean.Parser.Command.universe ||
  k == `Lean.Parser.Command.universes ||
  k == `Lean.Parser.Command.check ||
  k == `Lean.Parser.Command.set_option ||
  k == `Lean.Parser.Command.open

private partial def splitMutualPreamble (elems : Array Syntax) : Option (Array Syntax × Array Syntax) :=
  let rec loop (i : Nat) : Option (Array Syntax × Array Syntax) :=
    if h : i < elems.size then
      let elem := elems.get ⟨i, h⟩
      if isMutualPreambleCommand elem then
        loop (i+1)
      else if i == 0 then
        none -- `mutual` block does not contain any preamble commands
      else
        some (elems[0:i], elems[i:elems.size])
    else
      none -- a `mutual` block containing only preamble commands is not a valid `mutual` block
  loop 0

@[builtinMacro Lean.Parser.Command.mutual]
def expandMutualNamespace : Macro := fun stx => do
  let mut ns?      := none
  let mut elemsNew := #[]
  for elem in stx[1].getArgs do
    match ns?, expandDeclNamespace? elem with
    | _, none                         => elemsNew := elemsNew.push elem
    | none, some (ns, elem)           => ns? := some ns; elemsNew := elemsNew.push elem
    | some nsCurr, some (nsNew, elem) =>
```

```
      if nsCurr == nsNew then
        elemsNew := elemsNew.push elem
      else
        Macro.throwErrorAt elem s!"conflicting namespaces in mutual declaration, using namespace '{nsNew}', but used '{nsCurr}' in previ
  match ns? with
  | some ns =>
    let ns := mkIdentFrom stx ns
    let stxNew := stx.setArg 1 (mkNullNode elemsNew)
    `(namespace $ns:ident $stxNew end $ns:ident)
  | none => Macro.throwUnsupported

@[builtinMacro Lean.Parser.Command.mutual]
def expandMutualElement : Macro := fun stx => do
  let mut elemsNew := #[]
  let mut modified := false
  for elem in stx[1].getArgs do
    match (← expandMacro? elem) with
    | some elemNew => elemsNew := elemsNew.push elemNew; modified := true
    | none         => elemsNew := elemsNew.push elem
  if modified then
    pure $ stx.setArg 1 (mkNullNode elemsNew)
  else
    Macro.throwUnsupported

@[builtinMacro Lean.Parser.Command.mutual]
def expandMutualPreamble : Macro := fun stx =>
  match splitMutualPreamble stx[1].getArgs with
  | none => Macro.throwUnsupported
  | some (preamble, rest) => do
    let secCmd    ← `(section)
    let newMutual := stx.setArg 1 (mkNullNode rest)
    let endCmd    ← `(end)
    pure $ mkNullNode (#[secCmd] ++ preamble ++ #[newMutual] ++ #[endCmd])

@[builtinCommandElab «mutual»]
def elabMutual : CommandElab := fun stx => do
  if isMutualInductive stx then
    elabMutualInductive stx[1].getArgs
  else if isMutualDef stx then
    elabMutualDef stx[1].getArgs
  else
    throwError "invalid mutual block"

/- leading_parser "attribute " >> "[" >> sepBy1 (eraseAttr <|> Term.attrInstance) ", " >> "]" >> many1 ident -/
@[builtinCommandElab «attribute»] def elabAttr : CommandElab := fun stx => do
  let mut attrInsts := #[]
  let mut toErase := #[]
```

```
  for attrKindStx in stx[2].getSepArgs do
    if attrKindStx.getKind == ``Lean.Parser.Command.eraseAttr then
      let attrName := attrKindStx[1].getId.eraseMacroScopes
      unless isAttribute (← getEnv) attrName do
        throwError "unknown attribute [{attrName}]"
      toErase := toErase.push attrName
    else
      attrInsts := attrInsts.push attrKindStx
  let attrs ← elabAttrs attrInsts
  let idents := stx[4].getArgs
  for ident in idents do withRef ident <| liftTermElabM none do
    let declName ← resolveGlobalConstNoOverloadWithInfo ident
    Term.applyAttributes declName attrs
    for attrName in toErase do
      Attribute.erase declName attrName

def expandInitCmd (builtin : Bool) : Macro := fun stx =>
  let optHeader := stx[1]
  let doSeq     := stx[2]
  let attrId    := mkIdentFrom stx $ if builtin then `builtinInit else `init
  if optHeader.isNone then
    `(@[$attrId:ident]def initFn : IO Unit := do $doSeq)
  else
    let id   := optHeader[0]
    let type := optHeader[1][1]
    `(def initFn : IO $type := do $doSeq
      @[$attrId:ident initFn]constant $id : $type)

@[builtinMacro Lean.Parser.Command.«initialize»] def expandInitialize : Macro :=
  expandInitCmd (builtin := false)

@[builtinMacro Lean.Parser.Command.«builtin_initialize»] def expandBuiltinInitialize : Macro :=
  expandInitCmd (builtin := true)

end Lean.Elab.Command
::::::::::::::
Elab/DeclarationRange.lean
::::::::::::::
/-
Copyright (c) 2021 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.DeclarationRange
import Lean.Elab.Log
import Lean.Data.Lsp.Utf16

```

```
namespace Lean.Elab

def getDeclarationRange [Monad m] [MonadFileMap m] (stx : Syntax) : m DeclarationRange := do
  let fileMap ← getFileMap
  let pos    := stx.getPos?.getD 0
  let endPos := stx.getTailPos?.getD pos |> fileMap.toPosition
  let pos    := pos |> fileMap.toPosition
  return {
    pos          := pos
    charUtf16    := fileMap.leanPosToLspPos pos |>.character
    endPos       := endPos
    endCharUtf16 := fileMap.leanPosToLspPos endPos |>.character
  }

/--
  For most builtin declarations, the selection range is just its name, which is stored in the second position.
  Example:
  ```
  "def " >> declId >> optDeclSig >> declVal
  ```
  For instances, we use the whole header since the name is optional.
  This function converts the given `Syntax` into one that represents its "selection range".
-/
def getDeclarationSelectionRef (stx : Syntax) : Syntax :=
  if stx.getKind == ``Parser.Command.«instance» then
    stx.setArg 5 mkNullNode
  else
    stx[1]

/--
  Store the `range` and `selectionRange` for `declName` where `stx` is the whole syntax object decribing `declName`.
  This method is for the builtin declarations only.
  User-defined commands should use `Lean.addDeclarationRanges` to store this information for their commands. -/
def addDeclarationRanges [Monad m] [MonadEnv m] [MonadFileMap m] (declName : Name) (stx : Syntax) : m Unit := do
  if stx.getKind == ``Parser.Command.«example» then
    return ()
  else
    Lean.addDeclarationRanges declName {
      range          := (← getDeclarationRange stx)
      selectionRange := (← getDeclarationRange (getDeclarationSelectionRef stx))
    }

/-- Auxiliary method for recording ranges for auxiliary declarations (e.g., fields, nested declarations, etc. -/
def addAuxDeclarationRanges [Monad m] [MonadEnv m] [MonadFileMap m] (declName : Name) (stx : Syntax) (header : Syntax) : m Unit := do
  Lean.addDeclarationRanges declName {
    range          := (← getDeclarationRange stx)
    selectionRange := (← getDeclarationRange header)
```

```
  }

end Lean.Elab
::::::::::::::
Elab/DeclModifiers.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Modifiers
import Lean.DocString
import Lean.Elab.Attributes
import Lean.Elab.Exception
import Lean.Elab.DeclUtil

namespace Lean.Elab

def checkNotAlreadyDeclared {m} [Monad m] [MonadEnv m] [MonadError m] (declName : Name) : m Unit := do
  let env ← getEnv
  if env.contains declName then
    match privateToUserName? declName with
    | none          => throwError "'{declName}' has already been declared"
    | some declName => throwError "private declaration '{declName}' has already been declared"
  if env.contains (mkPrivateName env declName) then
    throwError "a private declaration '{declName}' has already been declared"
  match privateToUserName? declName with
  | none => pure ()
  | some declName =>
    if env.contains declName then
      throwError "a non-private declaration '{declName}' has already been declared"

inductive Visibility where
  | regular | «protected» | «private»
  deriving Inhabited

instance : ToString Visibility := ⟨fun
  | Visibility.regular     => "regular"
  | Visibility.«private»   => "private"
  | Visibility.«protected» => "protected"⟩

structure Modifiers where
  docString?      : Option String := none
  visibility      : Visibility := Visibility.regular
  isNoncomputable : Bool := false
  isPartial       : Bool := false
```

```
  isUnsafe        : Bool := false
  attrs           : Array Attribute := #[]
  deriving Inhabited

def Modifiers.isPrivate : Modifiers → Bool
  | { visibility := Visibility.private, .. } => true
  | _                                        => false

def Modifiers.isProtected : Modifiers → Bool
  | { visibility := Visibility.protected, .. } => true
  | _                                          => false

def Modifiers.addAttribute (modifiers : Modifiers) (attr : Attribute) : Modifiers :=
  { modifiers with attrs := modifiers.attrs.push attr }

instance : ToFormat Modifiers := ⟨fun m =>
  let components : List Format :=
    (match m.docString? with
     | some str => [f!"/--{str}-/"]
     | none     => [])
    ++ (match m.visibility with
     | Visibility.regular   => []
     | Visibility.protected => [f!"protected"]
     | Visibility.private   => [f!"private"])
    ++ (if m.isNoncomputable then [f!"noncomputable"] else [])
    ++ (if m.isPartial then [f!"partial"] else [])
    ++ (if m.isUnsafe then [f!"unsafe"] else [])
    ++ m.attrs.toList.map (fun attr => fmt attr)
  Format.bracket "{" (Format.joinSep components ("," ++ Format.line)) "}"⟩

instance : ToString Modifiers := ⟨toString ∘ format⟩

def expandOptDocComment? [Monad m] [MonadError m] (optDocComment : Syntax) : m (Option String) :=
  match optDocComment.getOptional? with
  | none   => pure none
  | some s => match s[1] with
    | Syntax.atom _ val => pure (some (val.extract 0 (val.bsize - 2)))
    | _                 => throwErrorAt s "unexpected doc string{indentD s[1]}"

section Methods

variable [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m]

def elabModifiers (stx : Syntax) : m Modifiers := do
  let docCommentStx := stx[0]
  let attrsStx      := stx[1]
  let visibilityStx := stx[2]
```

```
  let noncompStx    := stx[3]
  let unsafeStx     := stx[4]
  let partialStx    := stx[5]
  let docString? ← match docCommentStx.getOptional? with
    | none   => pure none
    | some s => match s[1] with
      | Syntax.atom _ val => pure (some (val.extract 0 (val.bsize - 2)))
      | _                 => throwErrorAt s "unexpected doc string{indentD s[1]}"
  let visibility ← match visibilityStx.getOptional? with
    | none   => pure Visibility.regular
    | some v =>
      let kind := v.getKind
      if kind == `Lean.Parser.Command.private then pure Visibility.private
      else if kind == `Lean.Parser.Command.protected then pure Visibility.protected
      else throwErrorAt v "unexpected visibility modifier"
  let attrs ← match attrsStx.getOptional? with
    | none       => pure #[]
    | some attrs => elabDeclAttrs attrs
  pure {
    docString?      := docString?,
    visibility      := visibility,
    isPartial       := !partialStx.isNone,
    isUnsafe        := !unsafeStx.isNone,
    isNoncomputable := !noncompStx.isNone,
    attrs           := attrs
  }

def applyVisibility (visibility : Visibility) (declName : Name) : m Name := do
  match visibility with
  | Visibility.private =>
    let env ← getEnv
    let declName := mkPrivateName env declName
    checkNotAlreadyDeclared declName
    pure declName
  | Visibility.protected =>
    checkNotAlreadyDeclared declName
    let env ← getEnv
    let env := addProtected env declName
    setEnv env
    pure declName
  | _ =>
    checkNotAlreadyDeclared declName
    pure declName

def mkDeclName (currNamespace : Name) (modifiers : Modifiers) (shortName : Name) : m (Name × Name) := do
  let name := (extractMacroScopes shortName).name
  unless name.isAtomic || isFreshInstanceName name do
```

```
    throwError "atomic identifier expected '{shortName}'"
  let declName := currNamespace ++ shortName
  let declName ← applyVisibility modifiers.visibility declName
  match modifiers.visibility with
  | Visibility.protected =>
    match currNamespace with
    | Name.str _ s _ => pure (declName, Name.mkSimple s ++ shortName)
    | _ => throwError "protected declarations must be in a namespace"
  | _ => pure (declName, shortName)

/-
  `declId` is of the form
  ```
  leading_parser ident >> optional (".{" >> sepBy1 ident ", " >> "}")
  ```
  but we also accept a single identifier to users to make macro writing more convenient .
-/
def expandDeclIdCore (declId : Syntax) : Name × Syntax :=
  if declId.isIdent then
    (declId.getId, mkNullNode)
  else
    let id             := declId[0].getId
    let optUnivDeclStx := declId[1]
    (id, optUnivDeclStx)

structure ExpandDeclIdResult where
  shortName  : Name
  declName   : Name
  levelNames : List Name

def expandDeclId (currNamespace : Name) (currLevelNames : List Name) (declId : Syntax) (modifiers : Modifiers) : m ExpandDeclIdResult :=
  -- ident >> optional (".{" >> sepBy1 ident ", " >> "}")
  let (shortName, optUnivDeclStx) := expandDeclIdCore declId
  let levelNames ←
    if optUnivDeclStx.isNone then
      pure currLevelNames
    else
      let extraLevels := optUnivDeclStx[1].getArgs.getEvenElems
      extraLevels.foldlM
        (fun levelNames idStx =>
          let id := idStx.getId
          if levelNames.elem id then
            withRef idStx $ throwAlreadyDeclaredUniverseLevel id
          else
            pure (id :: levelNames))
        currLevelNames
  let (declName, shortName) ← withRef declId $ mkDeclName currNamespace modifiers shortName
```

```
  addDocString' declName modifiers.docString?
  pure { shortName := shortName, declName := declName, levelNames := levelNames }

end Methods

end Lean.Elab
::::::::::::::
Elab/DeclUtil.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Meta.ExprDefEq

namespace Lean.Meta

def forallTelescopeCompatibleAux {α} (k : Array Expr → Expr → Expr → MetaM α) : Nat → Expr → Expr → Array Expr → MetaM α
  | 0, type₁, type₂, xs   => k xs type₁ type₂
  | i+1, type₁, type₂, xs => do
    let type₁ ← whnf type₁
    let type₂ ← whnf type₂
    match type₁, type₂ with
    | Expr.forallE n₁ d₁ b₁ c₁, Expr.forallE n₂ d₂ b₂ c₂ =>
      unless n₁ == n₂ do
        throwError "parameter name mismatch '{n₁}', expected '{n₂}'"
      unless (← isDefEq d₁ d₂) do
        throwError "parameter '{n₁}' {← mkHasTypeButIsExpectedMsg d₁ d₂}"
      unless c₁.binderInfo == c₂.binderInfo do
        throwError "binder annotation mismatch at parameter '{n₁}'"
      withLocalDecl n₁ c₁.binderInfo d₁ fun x =>
        let type₁ := b₁.instantiate1 x
        let type₂ := b₂.instantiate1 x
        forallTelescopeCompatibleAux k i type₁ type₂ (xs.push x)
    | _, _ => throwError "unexpected number of parameters"

/-- Given two forall-expressions `type₁` and `type₂`, ensure the first `numParams` parameters are compatible, and
    then execute `k` with the parameters and remaining types. -/
@[inline] def forallTelescopeCompatible {α m} [Monad m] [MonadControlT MetaM m] (type₁ type₂ : Expr) (numParams : Nat) (k : Array Expr →
  controlAt MetaM fun runInBase =>
    forallTelescopeCompatibleAux (fun xs type₁ type₂ => runInBase $ k xs type₁ type₂) numParams type₁ type₂ #[]

end Meta

namespace Elab

```

```lean
def expandOptDeclSig (stx : Syntax) : Syntax × Option Syntax :=
  -- many Term.bracketedBinder >> Term.optType
  let binders := stx[0]
  let optType := stx[1] -- optional (leading_parser " : " >> termParser)
  if optType.isNone then
    (binders, none)
  else
    let typeSpec := optType[0]
    (binders, some typeSpec[1])

def expandDeclSig (stx : Syntax) : Syntax × Syntax :=
  -- many Term.bracketedBinder >> Term.typeSpec
  let binders  := stx[0]
  let typeSpec := stx[1]
  (binders, typeSpec[1])

def mkFreshInstanceName (env : Environment) (nextIdx : Nat) : Name :=
  (env.mainModule ++ `_instance).appendIndexAfter nextIdx

def isFreshInstanceName (name : Name) : Bool :=
  match name with
  | Name.str _ s _ => "_instance".isPrefixOf s
  | _              => false

/--
  Sort the given list of `usedParams` using the following order:
  - If it is an explicit level `allUserParams`, then use user given order.
  - Otherwise, use lexicographical.

  Remark: `scopeParams` are the universe params introduced using the `universe` command. `allUserParams` contains
  the universe params introduced using the `universe` command *and* the `.{...}` notation.

  Remark: this function return an exception if there is an `u` not in `usedParams`, that is in `allUserParams` but not in `scopeParams`.

  Remark: `explicitParams` are in reverse declaration order. That is, the head is the last declared parameter. -/
def sortDeclLevelParams (scopeParams : List Name) (allUserParams : List Name) (usedParams : Array Name) : Except String (List Name) :=
  match allUserParams.find? $ fun u => !usedParams.contains u && !scopeParams.elem u with
  | some u => throw s!"unused universe parameter '{u}'"
  | none   =>
    let result := allUserParams.foldl (fun result levelName => if usedParams.elem levelName then levelName :: result else result) []
    let remaining := usedParams.filter (fun levelParam => !allUserParams.elem levelParam)
    let remaining := remaining.qsort Name.lt
    pure $ result ++ remaining.toList

end Lean.Elab
::::::::::::::
Elab/DefView.lean
```

```
:::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Std.ShareCommon
import Lean.Parser.Command
import Lean.Util.CollectLevelParams
import Lean.Util.FoldConsts
import Lean.Meta.CollectFVars
import Lean.Elab.Command
import Lean.Elab.SyntheticMVars
import Lean.Elab.Binders
import Lean.Elab.DeclUtil
namespace Lean.Elab

inductive DefKind where
  | «def» | «theorem» | «example» | «opaque» | «abbrev»
  deriving Inhabited

def DefKind.isTheorem : DefKind → Bool
  | «theorem» => true
  | _         => false

def DefKind.isDefOrAbbrevOrOpaque : DefKind → Bool
  | «def»    => true
  | «opaque» => true
  | «abbrev» => true
  | _        => false

def DefKind.isExample : DefKind → Bool
  | «example» => true
  | _         => false

structure DefView where
  kind          : DefKind
  ref           : Syntax
  modifiers     : Modifiers
  declId        : Syntax
  binders       : Syntax
  type?         : Option Syntax
  value         : Syntax
  deriving Inhabited

namespace Command

```

```
open Meta

def mkDefViewOfAbbrev (modifiers : Modifiers) (stx : Syntax) : DefView :=
  -- leading_parser "abbrev " >> declId >> optDeclSig >> declVal
  let (binders, type) := expandOptDeclSig (stx.getArg 2)
  let modifiers       := modifiers.addAttribute { name := `inline }
  let modifiers       := modifiers.addAttribute { name := `reducible }
  { ref := stx, kind := DefKind.abbrev, modifiers := modifiers,
    declId := stx.getArg 1, binders := binders, type? := type, value := stx.getArg 3 }

def mkDefViewOfDef (modifiers : Modifiers) (stx : Syntax) : DefView :=
  -- leading_parser "def " >> declId >> optDeclSig >> declVal
  let (binders, type) := expandOptDeclSig (stx.getArg 2)
  { ref := stx, kind := DefKind.def, modifiers := modifiers,
    declId := stx.getArg 1, binders := binders, type? := type, value := stx.getArg 3 }

def mkDefViewOfTheorem (modifiers : Modifiers) (stx : Syntax) : DefView :=
  -- leading_parser "theorem " >> declId >> declSig >> declVal
  let (binders, type) := expandDeclSig (stx.getArg 2)
  { ref := stx, kind := DefKind.theorem, modifiers := modifiers,
    declId := stx.getArg 1, binders := binders, type? := some type, value := stx.getArg 3 }

namespace MkInstanceName

-- Table for `mkInstanceName`
private def kindReplacements : NameMap String :=
  Std.RBMap.ofList [
    (``Parser.Term.depArrow, "DepArrow"),
    (``Parser.Term.«forall», "Forall"),
    (``Parser.Term.arrow, "Arrow"),
    (``Parser.Term.prop,  "Prop"),
    (``Parser.Term.sort,  "Sort"),
    (``Parser.Term.type,  "Type")
  ]

abbrev M := StateRefT String CommandElabM

def isFirst : M Bool :=
  return (← get) == ""

def append (str : String) : M Unit :=
  modify fun s => s ++ str

partial def collect (stx : Syntax) : M Unit := do
  match stx with
  | Syntax.node k args =>
    unless (← isFirst) do
```

```
      match kindReplacements.find? k with
      | some r => append r
      | none   => pure ()
    for arg in args do
      collect arg
  | Syntax.ident (preresolved := preresolved) .. =>
    unless preresolved.isEmpty && (← resolveGlobalName stx.getId).isEmpty do
      match stx.getId.eraseMacroScopes with
      | Name.str _ str _ =>
          if str[0].isLower then
            append str.capitalize
          else
            append str
      | _ => pure ()
  | _ => pure ()

def mkFreshInstanceName : CommandElabM Name := do
  let s ← get
  let idx := s.nextInstIdx
  modify fun s => { s with nextInstIdx := s.nextInstIdx + 1 }
  return Lean.Elab.mkFreshInstanceName s.env idx

partial def main (type : Syntax) : CommandElabM Name := do
  /- We use `expandMacros` to expand notation such as `x < y` into `LT.lt x y` -/
  let type ← liftMacroM <| expandMacros type
  let (_, str) ← collect type |>.run ""
  if str.isEmpty then
    mkFreshInstanceName
  else
    mkUnusedBaseName <| Name.mkSimple ("inst" ++ str)

end MkInstanceName

def mkDefViewOfConstant (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView := do
  -- leading_parser "constant " >> declId >> declSig >> optional declValSimple
  let (binders, type) := expandDeclSig (stx.getArg 2)
  let val ← match (stx.getArg 3).getOptional? with
    | some val => pure val
    | none     =>
      let val ← `(arbitrary)
      pure $ Syntax.node ``Parser.Command.declValSimple #[ mkAtomFrom stx ":=", val ]
  return {
    ref := stx, kind := DefKind.opaque, modifiers := modifiers,
    declId := stx.getArg 1, binders := binders, type? := some type, value := val
  }

def mkDefViewOfInstance (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView := do
```

```
  -- leading_parser Term.attrKind >> "instance " >> optNamedPrio >> optional declId >> declSig >> declVal
  let attrKind        ← toAttributeKind stx[0]
  let prio            ← liftMacroM <| expandOptNamedPrio stx[2]
  let attrStx         ← `(attr| instance $(quote prio):numLit)
  let (binders, type) := expandDeclSig stx[4]
  let modifiers       := modifiers.addAttribute { kind := attrKind, name := `instance, stx := attrStx }
  let declId ← match stx[3].getOptional? with
    | some declId => pure declId
    | none        =>
      let id ← MkInstanceName.main type
      pure <| Syntax.node ``Parser.Command.declId #[mkIdentFrom stx id, mkNullNode]
  return {
    ref := stx, kind := DefKind.def, modifiers := modifiers,
    declId := declId, binders := binders, type? := type, value := stx[5]
  }

def mkDefViewOfExample (modifiers : Modifiers) (stx : Syntax) : DefView :=
  -- leading_parser "example " >> declSig >> declVal
  let (binders, type) := expandDeclSig (stx.getArg 1)
  let id              := mkIdentFrom stx `_example
  let declId          := Syntax.node ``Parser.Command.declId #[id, mkNullNode]
  { ref := stx, kind := DefKind.example, modifiers := modifiers,
    declId := declId, binders := binders, type? := some type, value := stx.getArg 2 }

def isDefLike (stx : Syntax) : Bool :=
  let declKind := stx.getKind
  declKind == ``Parser.Command.«abbrev» ||
  declKind == ``Parser.Command.«def» ||
  declKind == ``Parser.Command.«theorem» ||
  declKind == ``Parser.Command.«constant» ||
  declKind == ``Parser.Command.«instance» ||
  declKind == ``Parser.Command.«example»

def mkDefView (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView :=
  let declKind := stx.getKind
  if declKind == ``Parser.Command.«abbrev» then
    pure $ mkDefViewOfAbbrev modifiers stx
  else if declKind == ``Parser.Command.«def» then
    pure $ mkDefViewOfDef modifiers stx
  else if declKind == ``Parser.Command.«theorem» then
    pure $ mkDefViewOfTheorem modifiers stx
  else if declKind == ``Parser.Command.«constant» then
    mkDefViewOfConstant modifiers stx
  else if declKind == ``Parser.Command.«instance» then
    mkDefViewOfInstance modifiers stx
  else if declKind == ``Parser.Command.«example» then
    pure $ mkDefViewOfExample modifiers stx
```

```
  else
    throwError "unexpected kind of definition"

builtin_initialize registerTraceClass `Elab.definition

end Command
end Lean.Elab
::::::::::::::
Elab/Deriving.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Deriving.Basic
import Lean.Elab.Deriving.Util
import Lean.Elab.Deriving.Inhabited
import Lean.Elab.Deriving.BEq
import Lean.Elab.Deriving.DecEq
import Lean.Elab.Deriving.Repr
import Lean.Elab.Deriving.FromToJson
import Lean.Elab.Deriving.SizeOf
import Lean.Elab.Deriving.Hashable
import Lean.Elab.Deriving.Ord
::::::::::::::
Elab/Do.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Term
import Lean.Elab.Binders
import Lean.Elab.Match
import Lean.Elab.Quotation.Util
import Lean.Parser.Do

namespace Lean.Elab.Term
open Lean.Parser.Term
open Meta

private def getDoSeqElems (doSeq : Syntax) : List Syntax :=
  if doSeq.getKind == `Lean.Parser.Term.doSeqBracketed then
    doSeq[1].getArgs.toList.map fun arg => arg[0]
  else if doSeq.getKind == `Lean.Parser.Term.doSeqIndent then
```

```
    doSeq[0].getArgs.toList.map fun arg => arg[0]
  else
    []

private def getDoSeq (doStx : Syntax) : Syntax :=
  doStx[1]

@[builtinTermElab liftMethod] def elabLiftMethod : TermElab := fun stx _ =>
  throwErrorAt stx "invalid use of `(<- ...)`, must be nested inside a 'do' expression"

/-- Return true if we should not lift `(<- ...)` actions nested in the syntax nodes with the given kind. -/
private def liftMethodDelimiter (k : SyntaxNodeKind) : Bool :=
  k == ``Lean.Parser.Term.do ||
  k == ``Lean.Parser.Term.doSeqIndent ||
  k == ``Lean.Parser.Term.doSeqBracketed ||
  k == ``Lean.Parser.Term.termReturn ||
  k == ``Lean.Parser.Term.termUnless ||
  k == ``Lean.Parser.Term.termTry ||
  k == ``Lean.Parser.Term.termFor

/-- Return true if we should generate an error message when lifting a method over this kind of syntax. -/
private def liftMethodForbiddenBinder (k : SyntaxNodeKind) : Bool :=
  k == ``Lean.Parser.Term.fun ||
  k == ``Lean.Parser.Term.matchAlts

private partial def hasLiftMethod : Syntax → Bool
  | Syntax.node k args =>
    if liftMethodDelimiter k then false
    -- NOTE: We don't check for lifts in quotations here, which doesn't break anything but merely makes this rare case a
    -- bit slower
    else if k == `Lean.Parser.Term.liftMethod then true
    else args.any hasLiftMethod
  | _ => false

structure ExtractMonadResult where
  m            : Expr
  α            : Expr
  hasBindInst  : Expr
  expectedType : Expr

private def mkIdBindFor (type : Expr) : TermElabM ExtractMonadResult := do
  let u ← getDecLevel type
  let id        := Lean.mkConst `Id [u]
  let idBindVal := Lean.mkConst `Id.hasBind [u]
  pure { m := id, hasBindInst := idBindVal, α := type, expectedType := mkApp id type }

private def extractBind (expectedType? : Option Expr) : TermElabM ExtractMonadResult := do
```

```
  match expectedType? with
  | none => throwError "invalid 'do' notation, expected type is not available"
  | some expectedType =>
    let type ← withReducible $ whnf expectedType
    if  type.getAppFn.isMVar then throwError "invalid 'do' notation, expected type is not available"
    match type with
    | Expr.app m α _ =>
      try
        let bindInstType ← mkAppM `Bind #[m]
        let bindInstVal  ← synthesizeInst bindInstType
        pure { m := m, hasBindInst := bindInstVal, α := α, expectedType := expectedType }
      catch _ =>
        mkIdBindFor type
    | _ => mkIdBindFor type

namespace Do

/- A `doMatch` alternative. `vars` is the array of variables declared by `patterns`. -/
structure Alt (σ : Type) where
  ref : Syntax
  vars : Array Name
  patterns : Syntax
  rhs : σ
  deriving Inhabited

/-
  Auxiliary datastructure for representing a `do` code block, and compiling "reassignments" (e.g., `x := x + 1`).
  We convert `Code` into a `Syntax` term representing the:
  - `do`-block, or
  - the visitor argument for the `forIn` combinator.

  We say the following constructors are terminals:
  - `break`:    for interrupting a `for x in s`
  - `continue`: for interrupting the current iteration of a `for x in s`
  - `return e`: for returning `e` as the result for the whole `do` computation block
  - `action a`: for executing action `a` as a terminal
  - `ite`:      if-then-else
  - `match`:    pattern matching
  - `jmp`       a goto to a join-point

  We say the terminals `break`, `continue`, `action`, and `return` are "exit points"

  Note that, `return e` is not equivalent to `action (pure e)`. Here is an example:
  ```
  def f (x : Nat) : IO Unit := do
  if x == 0 then
     return ()
```
```

```
  IO.println "hello"
  ```
  Executing `#eval f 0` will not print "hello". Now, consider
  ```
  def g (x : Nat) : IO Unit := do
  if x == 0 then
    pure ()
  IO.println "hello"
  ```
  The `if` statement is essentially a noop, and "hello" is printed when we execute `g 0`.

  - `decl` represents all declaration-like `doElem`s (e.g., `let`, `have`, `let rec`).
    The field `stx` is the actual `doElem`,
    `vars` is the array of variables declared by it, and `cont` is the next instruction in the `do` code block.
    `vars` is an array since we have declarations such as `let (a, b) := s`.

  - `reassign` is an reassignment-like `doElem` (e.g., `x := x + 1`).

  - `joinpoint` is a join point declaration: an auxiliary `let`-declaration used to represent the control-flow.

  - `seq a k` executes action `a`, ignores its result, and then executes `k`.
    We also store the do-elements `dbg_trace` and `assert!` as actions in a `seq`.

  A code block `C` is well-formed if
  - For every `jmp ref j as` in `C`, there is a `joinpoint j ps b k` and `jmp ref j as` is in `k`, and
    `ps.size == as.size` -/
inductive Code where
  | decl   (xs : Array Name) (doElem : Syntax) (k : Code)
  | reassign   (xs : Array Name) (doElem : Syntax) (k : Code)
  /- The Boolean value in `params` indicates whether we should use `(x : typeof! x)` when generating term Syntax or not -/
  | joinpoint   (name : Name) (params : Array (Name × Bool)) (body : Code) (k : Code)
  | seq   (action : Syntax) (k : Code)
  | action   (action : Syntax)
  | «break»   (ref : Syntax)
  | «continue»   (ref : Syntax)
  | «return»   (ref : Syntax) (val : Syntax)
  /- Recall that an if-then-else may declare a variable using `optIdent` for the branches `thenBranch` and `elseBranch`. We store the va
  | ite   (ref : Syntax) (h? : Option Name) (optIdent : Syntax) (cond : Syntax) (thenBranch : Code) (elseBranch : Code)
  | «match»   (ref : Syntax) (gen : Syntax) (discrs : Syntax) (optType : Syntax) (alts : Array (Alt Code))
  | jmp   (ref : Syntax) (jpName : Name) (args : Array Syntax)
  deriving Inhabited

/- A code block, and the collection of variables updated by it. -/
structure CodeBlock where
  code : Code
  uvars : NameSet := {} -- set of variables updated by `code`
```
```

```
private def nameSetToArray (s : NameSet) : Array Name :=
  s.fold (fun (xs : Array Name) x => xs.push x) #[]

private def varsToMessageData (vars : Array Name) : MessageData :=
  MessageData.joinSep (vars.toList.map fun n => MessageData.ofName (n.simpMacroScopes)) " "

partial def CodeBlocl.toMessageData (codeBlock : CodeBlock) : MessageData :=
  let us := MessageData.ofList $ (nameSetToArray codeBlock.uvars).toList.map MessageData.ofName
  let rec loop : Code → MessageData
    | Code.decl xs _ k            => m!"let {varsToMessageData xs} := ...\n{loop k}"
    | Code.reassign xs _ k        => m!"{varsToMessageData xs} := ...\n{loop k}"
    | Code.joinpoint n ps body k  => m!"let {n.simpMacroScopes} {varsToMessageData (ps.map Prod.fst)} := {indentD (loop body)}\n{loop k}
    | Code.seq e k                => m!"{e}\n{loop k}"
    | Code.action e               => e
    | Code.ite _ _ _ c t e        => m!"if {c} then {indentD (loop t)}\nelse{loop e}"
    | Code.jmp _ j xs             => m!"jmp {j.simpMacroScopes} {xs.toList}"
    | Code.«break» _              => m!"break {us}"
    | Code.«continue» _           => m!"continue {us}"
    | Code.«return» _ v           => m!"return {v} {us}"
    | Code.«match» _ _ ds t alts  =>
      m!"match {ds} with"
      ++ alts.foldl (init := m!"") fun acc alt => acc ++ m!"\n| {alt.patterns} => {loop alt.rhs}"
  loop codeBlock.code

/- Return true if the give code contains an exit point that satisfies `p` -/
@[inline] partial def hasExitPointPred (c : Code) (p : Code → Bool) : Bool :=
  let rec @[specialize] loop : Code → Bool
    | Code.decl _ _ k           => loop k
    | Code.reassign _ _ k       => loop k
    | Code.joinpoint _ _ b k    => loop b || loop k
    | Code.seq _ k              => loop k
    | Code.ite _ _ _ _ t e      => loop t || loop e
    | Code.«match» _ _ _ _ alts => alts.any (loop ·.rhs)
    | Code.jmp _ _ _            => false
    | c                         => p c
  loop c

def hasExitPoint (c : Code) : Bool :=
  hasExitPointPred c fun c => true

def hasReturn (c : Code) : Bool :=
  hasExitPointPred c fun
    | Code.«return» _ _ => true
    | _ => false

def hasTerminalAction (c : Code) : Bool :=
  hasExitPointPred c fun
```

```
    | Code.«action» _ => true
    | _ => false

def hasBreakContinue (c : Code) : Bool :=
  hasExitPointPred c fun
    | Code.«break» _    => true
    | Code.«continue» _ => true
    | _ => false

def hasBreakContinueReturn (c : Code) : Bool :=
  hasExitPointPred c fun
    | Code.«break» _    => true
    | Code.«continue» _ => true
    | Code.«return» _ _ => true
    | _ => false

def mkAuxDeclFor {m} [Monad m] [MonadQuotation m] (e : Syntax) (mkCont : Syntax → m Code) : m Code := withRef e <| withFreshMacroScope d
  let y ← `(y)
  let yName := y.getId
  let doElem ← `(doElem| let y ← $e:term)
  -- Add elaboration hint for producing sane error message
  let y ← `(ensureExpectedType% "type mismatch, result value" $y)
  let k ← mkCont y
  pure $ Code.decl #[yName] doElem k

/- Convert `action _ e` instructions in `c` into `let y ← e; jmp _ jp (xs y)`. -/
partial def convertTerminalActionIntoJmp (code : Code) (jp : Name) (xs : Array Name) : MacroM Code :=
  let rec loop : Code → MacroM Code
    | Code.decl xs stx k           => do Code.decl xs stx (← loop k)
    | Code.reassign xs stx k       => do Code.reassign xs stx (← loop k)
    | Code.joinpoint n ps b k      => do Code.joinpoint n ps (← loop b) (← loop k)
    | Code.seq e k                 => do Code.seq e (← loop k)
    | Code.ite ref x? h c t e      => do Code.ite ref x? h c (← loop t) (← loop e)
    | Code.«match» ref g ds t alts => do Code.«match» ref g ds t (← alts.mapM fun alt => do pure { alt with rhs := (← loop alt.rhs) })
    | Code.action e                => mkAuxDeclFor e fun y =>
      let ref := e
      -- We jump to `jp` with xs **and** y
      let jmpArgs := xs.map $ mkIdentFrom ref
      let jmpArgs := jmpArgs.push y
      pure $ Code.jmp ref jp jmpArgs
    | c                            => pure c
  loop code

structure JPDecl where
  name : Name
  params : Array (Name × Bool)
  body : Code
```

```
def attachJP (jpDecl : JPDecl) (k : Code) : Code :=
  Code.joinpoint jpDecl.name jpDecl.params jpDecl.body k

def attachJPs (jpDecls : Array JPDecl) (k : Code) : Code :=
  jpDecls.foldr attachJP k

def mkFreshJP (ps : Array (Name × Bool)) (body : Code) : TermElabM JPDecl := do
  let ps ←
    if ps.isEmpty then
      let y ← mkFreshUserName `y
      pure #[(y, false)]
    else
      pure ps
  -- Remark: the compiler frontend implemented in C++ currently detects jointpoints created by
  -- the "do" notation by testing the name. See hack at method `visit_let` at `lcnf.cpp`
  -- We will remove this hack when we re-implement the compiler frontend in Lean.
  let name ← mkFreshUserName `_do_jp
  pure { name := name, params := ps, body := body }

def mkFreshJP' (xs : Array Name) (body : Code) : TermElabM JPDecl :=
  mkFreshJP (xs.map fun x => (x, true)) body

def addFreshJP (ps : Array (Name × Bool)) (body : Code) : StateRefT (Array JPDecl) TermElabM Name := do
  let jp ← mkFreshJP ps body
  modify fun (jps : Array JPDecl) => jps.push jp
  pure jp.name

def insertVars (rs : NameSet) (xs : Array Name) : NameSet :=
  xs.foldl (·.insert ·) rs

def eraseVars (rs : NameSet) (xs : Array Name) : NameSet :=
  xs.foldl (·.erase ·) rs

def eraseOptVar (rs : NameSet) (x? : Option Name) : NameSet :=
  match x? with
  | none   => rs
  | some x => rs.insert x

/- Create a new jointpoint for `c`, and jump to it with the variables `rs` -/
def mkSimpleJmp (ref : Syntax) (rs : NameSet) (c : Code) : StateRefT (Array JPDecl) TermElabM Code := do
  let xs := nameSetToArray rs
  let jp ← addFreshJP (xs.map fun x => (x, true)) c
  if xs.isEmpty then
    let unit ← `(Unit.unit)
    return Code.jmp ref jp #[unit]
  else
```

```
    return Code.jmp ref jp (xs.map $ mkIdentFrom ref)

/- Create a new joinpoint that takes `rs` and `val` as arguments. `val` must be syntax representing a pure value.
   The body of the joinpoint is created using `mkJPBody yFresh`, where `yFresh`
   is a fresh variable created by this method. -/
def mkJmp (ref : Syntax) (rs : NameSet) (val : Syntax) (mkJPBody : Syntax → MacroM Code) : StateRefT (Array JPDecl) TermElabM Code := do
  let xs := nameSetToArray rs
  let args := xs.map $ mkIdentFrom ref
  let args := args.push val
  let yFresh ← mkFreshUserName `y
  let ps := xs.map fun x => (x, true)
  let ps := ps.push (yFresh, false)
  let jpBody ← liftMacroM $ mkJPBody (mkIdentFrom ref yFresh)
  let jp ← addFreshJP ps jpBody
  pure $ Code.jmp ref jp args

/- `pullExitPointsAux rs c` auxiliary method for `pullExitPoints`, `rs` is the set of update variable in the current path.  -/
partial def pullExitPointsAux : NameSet → Code → StateRefT (Array JPDecl) TermElabM Code
  | rs, Code.decl xs stx k           => do Code.decl xs stx (← pullExitPointsAux (eraseVars rs xs) k)
  | rs, Code.reassign xs stx k       => do Code.reassign xs stx (← pullExitPointsAux (insertVars rs xs) k)
  | rs, Code.joinpoint j ps b k      => do Code.joinpoint j ps (← pullExitPointsAux rs b) (← pullExitPointsAux rs k)
  | rs, Code.seq e k                 => do Code.seq e (← pullExitPointsAux rs k)
  | rs, Code.ite ref x? o c t e      => do Code.ite ref x? o c (← pullExitPointsAux (eraseOptVar rs x?) t) (← pullExitPointsAux (eraseOp
  | rs, Code.«match» ref g ds t alts => do
    Code.«match» ref g ds t (← alts.mapM fun alt => do pure { alt with rhs := (← pullExitPointsAux (eraseVars rs alt.vars) alt.rhs) })
  | rs, c@(Code.jmp _ _ _)           => pure c
  | rs, Code.«break» ref             => mkSimpleJmp ref rs (Code.«break» ref)
  | rs, Code.«continue» ref          => mkSimpleJmp ref rs (Code.«continue» ref)
  | rs, Code.«return» ref val        => mkJmp ref rs val (fun y => pure $ Code.«return» ref y)
  | rs, Code.action e                =>
    -- We use `mkAuxDeclFor` because `e` is not pure.
    mkAuxDeclFor e fun y =>
      let ref := e
      mkJmp ref rs y (fun yFresh => do pure $ Code.action (← `(Pure.pure $yFresh)))

/-
Auxiliary operation for adding new variables to the collection of updated variables in a CodeBlock.
When a new variable is not already in the collection, but is shadowed by some declaration in `c`,
we create auxiliary join points to make sure we preserve the semantics of the code block.
Example: suppose we have the code block `print x; let x := 10; return x`. And we want to extend it
with the reassignment `x := x + 1`. We first use `pullExitPoints` to create
```
let jp (x!1) := return x!1;
print x;
let x := 10;
jmp jp x
```
```

```
and then we add the reassignment
```
x := x + 1
let jp (x!1) := return x!1;
print x;
let x := 10;
jmp jp x
```
Note that we created a fresh variable `x!1` to avoid accidental name capture.
As another example, consider
```
print x;
let x := 10
y := y + 1;
return x;
```
We transform it into
```
let jp (y x!1) := return x!1;
print x;
let x := 10
y := y + 1;
jmp jp y x
```
and then we add the reassignment as in the previous example.
We need to include `y` in the jump, because each exit point is implicitly returning the set of
update variables.

We implement the method as follows. Let `us` be `c.uvars`, then
1- for each `return _ y` in `c`, we create a join point
  `let j (us y!1) := return y!1`
   and replace the `return _ y` with `jmp us y`
2- for each `break`, we create a join point
  `let j (us) := break`
   and replace the `break` with `jmp us`.
3- Same as 2 for `continue`.
-/
def pullExitPoints (c : Code) : TermElabM Code := do
  if hasExitPoint c then
    let (c, jpDecls) ← (pullExitPointsAux {} c).run #[]
    pure $ attachJPs jpDecls c
  else
    pure c

partial def extendUpdatedVarsAux (c : Code) (ws : NameSet) : TermElabM Code :=
  let rec update : Code → TermElabM Code
    | Code.joinpoint j ps b k          => do Code.joinpoint j ps (← update b) (← update k)
```

```
    | Code.seq e k                      => do Code.seq e (← update k)
    | c@(Code.«match» ref g ds t alts) => do
      if alts.any fun alt => alt.vars.any fun x => ws.contains x then
        -- If a pattern variable is shadowing a variable in ws, we `pullExitPoints`
        pullExitPoints c
      else
        Code.«match» ref g ds t (← alts.mapM fun alt => do pure { alt with rhs := (← update alt.rhs) })
    | Code.ite ref none o c t e => do Code.ite ref none o c (← update t) (← update e)
    | c@(Code.ite ref (some h) o cond t e) => do
      if ws.contains h then
        -- if the `h` at `if h:c then t else e` shadows a variable in `ws`, we `pullExitPoints`
        pullExitPoints c
      else
        Code.ite ref (some h) o cond (← update t) (← update e)
    | Code.reassign xs stx k => do Code.reassign xs stx (← update k)
    | c@(Code.decl xs stx k) => do
      if xs.any fun x => ws.contains x then
        -- One the declared variables is shadowing a variable in `ws`
        pullExitPoints c
      else
        Code.decl xs stx (← update k)
    | c => pure c
  update c

/-
Extend the set of updated variables. It assumes `ws` is a super set of `c.uvars`.
We **cannot** simply update the field `c.uvars`, because `c` may have shadowed some variable in `ws`.
See discussion at `pullExitPoints`.
-/
partial def extendUpdatedVars (c : CodeBlock) (ws : NameSet) : TermElabM CodeBlock := do
  if ws.any fun x => !c.uvars.contains x then
    -- `ws` contains a variable that is not in `c.uvars`, but in `c.dvars` (i.e., it has been shadowed)
    pure { code := (← extendUpdatedVarsAux c.code ws), uvars := ws }
  else
    pure { c with uvars := ws }

private def union (s₁ s₂ : NameSet) : NameSet :=
  s₁.fold (·.insert ·) s₂

/-
Given two code blocks `c₁` and `c₂`, make sure they have the same set of updated variables.
Let `ws` the union of the updated variables in `c₁` and `c₂`.
We use `extendUpdatedVars c₁ ws` and `extendUpdatedVars c₂ ws`
-/
def homogenize (c₁ c₂ : CodeBlock) : TermElabM (CodeBlock × CodeBlock) := do
  let ws := union c₁.uvars c₂.uvars
  let c₁ ← extendUpdatedVars c₁ ws
```

```
  let c₂ ← extendUpdatedVars c₂ ws
  pure (c₁, c₂)

/-
Extending code blocks with variable declarations: `let x : t := v` and `let x : t ← v`.
We remove `x` from the collection of updated varibles.
Remark: `stx` is the syntax for the declaration (e.g., `letDecl`), and `xs` are the variables
declared by it. It is an array because we have let-declarations that declare multiple variables.
Example: `let (x, y) := t`
-/
def mkVarDeclCore (xs : Array Name) (stx : Syntax) (c : CodeBlock) : CodeBlock := {
  code := Code.decl xs stx c.code,
  uvars := eraseVars c.uvars xs
}

/-
Extending code blocks with reassignments: `x : t := v` and `x : t ← v`.
Remark: `stx` is the syntax for the declaration (e.g., `letDecl`), and `xs` are the variables
declared by it. It is an array because we have let-declarations that declare multiple variables.
Example: `(x, y) ← t`
-/
def mkReassignCore (xs : Array Name) (stx : Syntax) (c : CodeBlock) : TermElabM CodeBlock := do
  let us := c.uvars
  let ws := insertVars us xs
  -- If `xs` contains a new updated variable, then we must use `extendUpdatedVars`.
  -- See discussion at `pullExitPoints`
  let code ← if xs.any fun x => !us.contains x then extendUpdatedVarsAux c.code ws else pure c.code
  pure { code := Code.reassign xs stx code, uvars := ws }

def mkSeq (action : Syntax) (c : CodeBlock) : CodeBlock :=
  { c with code := Code.seq action c.code }

def mkTerminalAction (action : Syntax) : CodeBlock :=
  { code := Code.action action }

def mkReturn (ref : Syntax) (val : Syntax) : CodeBlock :=
  { code := Code.«return» ref val }

def mkBreak (ref : Syntax) : CodeBlock :=
  { code := Code.«break» ref }

def mkContinue (ref : Syntax) : CodeBlock :=
  { code := Code.«continue» ref }

def mkIte (ref : Syntax) (optIdent : Syntax) (cond : Syntax) (thenBranch : CodeBlock) (elseBranch : CodeBlock) : TermElabM CodeBlock :=
  let x? := if optIdent.isNone then none else some optIdent[0].getId
  let (thenBranch, elseBranch) ← homogenize thenBranch elseBranch
```

```
  pure {
    code  := Code.ite ref x? optIdent cond thenBranch.code elseBranch.code,
    uvars := thenBranch.uvars,
  }

private def mkUnit : MacroM Syntax :=
  `((() : PUnit))

private def mkPureUnit : MacroM Syntax :=
  `(pure PUnit.unit)

def mkPureUnitAction : MacroM CodeBlock := do
  mkTerminalAction (← mkPureUnit)

def mkUnless (cond : Syntax) (c : CodeBlock) : MacroM CodeBlock := do
  let thenBranch ← mkPureUnitAction
  pure { c with code := Code.ite (← getRef) none mkNullNode cond thenBranch.code c.code }

def mkMatch (ref : Syntax) (genParam : Syntax) (discrs : Syntax) (optType : Syntax) (alts : Array (Alt CodeBlock)) : TermElabM CodeBlock
  -- nary version of homogenize
  let ws := alts.foldl (union · ·.rhs.uvars) {}
  let alts ← alts.mapM fun alt => do
    let rhs ← extendUpdatedVars alt.rhs ws
    pure { ref := alt.ref, vars := alt.vars, patterns := alt.patterns, rhs := rhs.code : Alt Code }
  pure { code := Code.«match» ref genParam discrs optType alts, uvars := ws }

/- Return a code block that executes `terminal` and then `k` with the value produced by `terminal`.
   This method assumes `terminal` is a terminal -/
def concat (terminal : CodeBlock) (kRef : Syntax) (y? : Option Name) (k : CodeBlock) : TermElabM CodeBlock := do
  unless hasTerminalAction terminal.code do
    throwErrorAt kRef "'do' element is unreachable"
  let (terminal, k) ← homogenize terminal k
  let xs := nameSetToArray k.uvars
  let y ← match y? with | some y => pure y | none => mkFreshUserName `y
  let ps := xs.map fun x => (x, true)
  let ps := ps.push (y, false)
  let jpDecl ← mkFreshJP ps k.code
  let jp := jpDecl.name
  let terminal ← liftMacroM $ convertTerminalActionIntoJmp terminal.code jp xs
  pure { code  := attachJP jpDecl terminal, uvars := k.uvars }

def getLetIdDeclVar (letIdDecl : Syntax) : Name :=
  letIdDecl[0].getId

def getPatternVarNames (pvars : Array PatternVar) : Array Name :=
  pvars.filterMap fun
    | PatternVar.localVar x => some x
```

```
    | _ => none

-- support both regular and syntax match
def getPatternVarsEx (pattern : Syntax) : TermElabM (Array Name) :=
  getPatternVarNames <$> getPatternVars pattern <|>
  Array.map Syntax.getId <$> Quotation.getPatternVars pattern

def getPatternsVarsEx (patterns : Array Syntax) : TermElabM (Array Name) :=
  getPatternVarNames <$> getPatternsVars patterns <|>
  Array.map Syntax.getId <$> Quotation.getPatternsVars patterns

def getLetPatDeclVars (letPatDecl : Syntax) : TermElabM (Array Name) := do
  let pattern := letPatDecl[0]
  getPatternVarsEx pattern

def getLetEqnsDeclVar (letEqnsDecl : Syntax) : Name :=
  letEqnsDecl[0].getId

def getLetDeclVars (letDecl : Syntax) : TermElabM (Array Name) := do
  let arg := letDecl[0]
  if arg.getKind == `Lean.Parser.Term.letIdDecl then
    pure #[getLetIdDeclVar arg]
  else if arg.getKind == `Lean.Parser.Term.letPatDecl then
    getLetPatDeclVars arg
  else if arg.getKind == `Lean.Parser.Term.letEqnsDecl then
    pure #[getLetEqnsDeclVar arg]
  else
    throwError "unexpected kind of let declaration"

def getDoLetVars (doLet : Syntax) : TermElabM (Array Name) :=
  -- leading_parser "let " >> optional "mut " >> letDecl
  getLetDeclVars doLet[2]

def getDoHaveVar (doHave : Syntax) : Name :=
  /-
    `leading_parser "have " >> Term.haveDecl`
    where
    ```
    haveDecl := leading_parser optIdent >> termParser >> (haveAssign <|> fromTerm <|> byTactic)
    optIdent := optional (try (ident >> " : "))

    ```
  -/
  let optIdent := doHave[1][0]
  if optIdent.isNone then
    `this
  else
```

```
    optIdent[0].getId

def getDoLetRecVars (doLetRec : Syntax) : TermElabM (Array Name) := do
  -- letRecDecls is an array of `(group (optional attributes >> letDecl))`
  let letRecDecls := doLetRec[1][0].getSepArgs
  let letDecls := letRecDecls.map fun p => p[2]
  let mut allVars := #[]
  for letDecl in letDecls do
    let vars ← getLetDeclVars letDecl
    allVars := allVars ++ vars
  pure allVars

-- ident >> optType >> leftArrow >> termParser
def getDoIdDeclVar (doIdDecl : Syntax) : Name :=
  doIdDecl[0].getId

-- termParser >> leftArrow >> termParser >> optional (" | " >> termParser)
def getDoPatDeclVars (doPatDecl : Syntax) : TermElabM (Array Name) := do
  let pattern := doPatDecl[0]
  getPatternVarsEx pattern

-- leading_parser "let " >> optional "mut " >> (doIdDecl <|> doPatDecl)
def getDoLetArrowVars (doLetArrow : Syntax) : TermElabM (Array Name) := do
  let decl := doLetArrow[2]
  if decl.getKind == `Lean.Parser.Term.doIdDecl then
    pure #[getDoIdDeclVar decl]
  else if decl.getKind == `Lean.Parser.Term.doPatDecl then
    getDoPatDeclVars decl
  else
    throwError "unexpected kind of 'do' declaration"

def getDoReassignVars (doReassign : Syntax) : TermElabM (Array Name) := do
  let arg := doReassign[0]
  if arg.getKind == `Lean.Parser.Term.letIdDecl then
    pure #[getLetIdDeclVar arg]
  else if arg.getKind == `Lean.Parser.Term.letPatDecl then
    getLetPatDeclVars arg
  else
    throwError "unexpected kind of reassignment"

def mkDoSeq (doElems : Array Syntax) : Syntax :=
  mkNode `Lean.Parser.Term.doSeqIndent #[mkNullNode $ doElems.map fun doElem => mkNullNode #[doElem, mkNullNode]]

def mkSingletonDoSeq (doElem : Syntax) : Syntax :=
  mkDoSeq #[doElem]

/-
```

```
  If the given syntax is a `doIf`, return an equivalente `doIf` that has an `else` but no `else if`s or `if let`s.  -/
private def expandDoIf? (stx : Syntax) : MacroM (Option Syntax) := match stx with
  | `(doElem|if $p:doIfProp then $t else $e) => pure none
  | `(doElem|if%$i $cond:doIfCond then $t $[else if%$is $conds:doIfCond then $ts]* $[else $e?]?) => withRef stx do
    let mut e      := e?.getD (← `(doSeq|pure PUnit.unit))
    let mut eIsSeq := true
    for (i, cond, t) in Array.zip (is.reverse.push i) (Array.zip (conds.reverse.push cond) (ts.reverse.push t)) do
      e ← if eIsSeq then e else `(doSeq|$e:doElem)
      e ← withRef cond <| match cond with
        | `(doIfCond|let $pat := $d) => `(doElem| match%$i $d:term with | $pat:term => $t | _ => $e)
        | `(doIfCond|let $pat ← $d)  => `(doElem| match%$i ← $d    with | $pat:term => $t | _ => $e)
        | `(doIfCond|$cond:doIfProp) => `(doElem| if%$i $cond:doIfProp then $t else $e)
        | _                          => `(doElem| if%$i $(Syntax.missing) then $t else $e)
      eIsSeq := false
    return some e
  | _ => pure none

structure DoIfView where
  ref        : Syntax
  optIdent   : Syntax
  cond       : Syntax
  thenBranch : Syntax
  elseBranch : Syntax

/- This method assumes `expandDoIf?` is not applicable. -/
private def mkDoIfView (doIf : Syntax) : MacroM DoIfView := do
  pure {
    ref        := doIf,
    optIdent   := doIf[1][0],
    cond       := doIf[1][1],
    thenBranch := doIf[3],
    elseBranch := doIf[5][1]
  }

/-
We use `MProd` instead of `Prod` to group values when expanding the
`do` notation. `MProd` is a universe monomorphic product.
The motivation is to generate simpler universe constraints in code
that was not written by the user.
Note that we are not restricting the macro power since the
`Bind.bind` combinator already forces values computed by monadic
actions to be in the same universe.
-/
private def mkTuple (elems : Array Syntax) : MacroM Syntax := do
  if elems.size == 0 then
    mkUnit
  else if elems.size == 1 then
```

```
    pure elems[0]
  else
    (elems.extract 0 (elems.size - 1)).foldrM
      (fun elem tuple => `(MProd.mk $elem $tuple))
      (elems.back)

/- Return `some action` if `doElem` is a `doExpr <action>`-/
def isDoExpr? (doElem : Syntax) : Option Syntax :=
  if doElem.getKind == `Lean.Parser.Term.doExpr then
    some doElem[0]
  else
    none

/--
  Given `uvars := #[a_1, ..., a_n, a_{n+1}]` construct term
  ```
  let a_1   := x.1
  let x   := x.2
  let a_2   := x.1
  let x   := x.2
  ...
  let a_n   := x.1
  let a_{n+1} := x.2
  body
  ```
  Special cases
  - `uvars := #[]` => `body`
  - `uvars := #[a]` => `let a := x; body`


  We use this method when expanding the `for-in` notation.
-/
private def destructTuple (uvars : Array Name) (x : Syntax) (body : Syntax) : MacroM Syntax := do
  if uvars.size == 0 then
    return body
  else if uvars.size == 1 then
    `(let $(← mkIdentFromRef uvars[0]):ident := $x; $body)
  else
    destruct uvars.toList x body
where
  destruct (as : List Name) (x : Syntax) (body : Syntax) : MacroM Syntax := do
    match as with
      | [a, b]  => `(let $(← mkIdentFromRef a):ident := $x.1; let $(← mkIdentFromRef b):ident := $x.2; $body)
      | a :: as => withFreshMacroScope do
        let rest ← destruct as (← `(x)) body
        `(let $(← mkIdentFromRef a):ident := $x.1; let x := $x.2; $rest)
      | _ => unreachable!
```

```
/-
The procedure `ToTerm.run` converts a `CodeBlock` into a `Syntax` term.
We use this method to convert
1- The `CodeBlock` for a root `do ...` term into a `Syntax` term. This kind of
   `CodeBlock` never contains `break` nor `continue`. Moreover, the collection
   of updated variables is not packed into the result.
   Thus, we have two kinds of exit points
     - `Code.action e` which is converted into `e`
     - `Code.return _ e` which is converted into `pure e`

   We use `Kind.regular` for this case.

2- The `CodeBlock` for `b` at `for x in xs do b`. In this case, we need to generate
   a `Syntax` term representing a function for the `xs.forIn` combinator.

   a) If `b` contain a `Code.return _ a` exit point. The generated `Syntax` term
      has type `m (ForInStep (Option α × σ))`, where `a : α`, and the `σ` is the type
      of the tuple of variables reassigned by `b`.
      We use `Kind.forInWithReturn` for this case

   b) If `b` does not contain a `Code.return _ a` exit point. Then, the generated
      `Syntax` term has type `m (ForInStep σ)`.
      We use `Kind.forIn` for this case.

3- The `CodeBlock` `c` for a `do` sequence nested in a monadic combinator (e.g., `MonadExcept.tryCatch`).

   The generated `Syntax` term for `c` must inform whether `c` "exited" using `Code.action`, `Code.return`,
   `Code.break` or `Code.continue`. We use the auxiliary types `DoResult`s for storing this information.
   For example, the auxiliary type `DoResultPBC α σ` is used for a code block that exits with `Code.action`,
   **and** `Code.break`/`Code.continue`, `α` is the type of values produced by the exit `action`, and
   `σ` is the type of the tuple of reassigned variables.
   The type `DoResult α β σ` is usedf for code blocks that exit with
   `Code.action`, `Code.return`, **and** `Code.break`/`Code.continue`, `β` is the type of the returned values.
   We don't use `DoResult α β σ` for all cases because:

      a) The elaborator would not be able to infer all type parameters without extra annotations. For example,
         if the code block does not contain `Code.return _ _`, the elaborator will not be able to infer `β`.

      b) We need to pattern match on the result produced by the combinator (e.g., `MonadExcept.tryCatch`),
         but we don't want to consider "unreachable" cases.

   We do not distinguish between cases that contain `break`, but not `continue`, and vice versa.

   When listing all cases, we use `a` to indicate the code block contains `Code.action _`, `r` for `Code.return _ _`,
   and `b/c` for a code block that contains `Code.break _` or `Code.continue _`.

```

```
  - `a`: `Kind.regular`, type `m (α × σ)`

  - `r`: `Kind.regular`, type `m (α × σ)`
          Note that the code that pattern matches on the result will behave differently in this case.
          It produces `return a` for this case, and `pure a` for the previous one.

  - `b/c`: `Kind.nestedBC`, type `m (DoResultBC σ)`

  - `a` and `r`:   `Kind.nestedPR`, type `m (DoResultPR α β σ)`

  - `a` and `bc`:  `Kind.nestedSBC`, type `m (DoResultSBC α σ)`

  - `r` and `bc`:  `Kind.nestedSBC`, type `m (DoResultSBC α σ)`
        Again the code that pattern matches on the result will behave differently in this case and
        the previous one. It produces `return a` for the constructor `DoResultSPR.pureReturn a u` for
        this case, and `pure a` for the previous case.

  - `a`, `r`, `b/c`: `Kind.nestedPRBC`, type type `m (DoResultPRBC α β σ)`

Here is the recipe for adding new combinators with nested `do`s.
Example: suppose we want to support `repeat doSeq`. Assuming we have `repeat : m α → m α`
1- Convert `doSeq` into `codeBlock : CodeBlock`
2- Create term `term` using `mkNestedTerm code m uvars a r bc` where
   `code` is `codeBlock.code`, `uvars` is an array containing `codeBlock.uvars`,
   `m` is a `Syntax` representing the Monad, and
   `a` is true if `code` contains `Code.action _`,
   `r` is true if `code` contains `Code.return _ _`,
   `bc` is true if `code` contains `Code.break _` or `Code.continue _`.

   Remark: for combinators such as `repeat` that take a single `doSeq`, all
   arguments, but `m`, are extracted from `codeBlock`.
3- Create the term `repeat $term`
4- and then, convert it into a `doSeq` using `matchNestedTermResult ref (repeat $term) uvsar a r bc`

-/
namespace ToTerm

inductive Kind where
  | regular
  | forIn
  | forInWithReturn
  | nestedBC
  | nestedPR
  | nestedSBC
  | nestedPRBC

instance : Inhabited Kind := ⟨Kind.regular⟩
```

```

def Kind.isRegular : Kind → Bool
  | Kind.regular => true
  | _            => false

structure Context where
  m     : Syntax -- Syntax to reference the monad associated with the do notation.
  uvars : Array Name
  kind  : Kind

abbrev M := ReaderT Context MacroM

def mkUVarTuple : M Syntax := do
  let ctx ← read
  let uvarIdents ← ctx.uvars.mapM mkIdentFromRef
  mkTuple uvarIdents

def returnToTerm (val : Syntax) : M Syntax := do
  let ctx ← read
  let u ← mkUVarTuple
  match ctx.kind with
  | Kind.regular         => if ctx.uvars.isEmpty then `(Pure.pure $val) else `(Pure.pure (MProd.mk $val $u))
  | Kind.forIn           => `(Pure.pure (ForInStep.done $u))
  | Kind.forInWithReturn => `(Pure.pure (ForInStep.done (MProd.mk (some $val) $u)))
  | Kind.nestedBC        => unreachable!
  | Kind.nestedPR        => `(Pure.pure (DoResultPR.«return» $val $u))
  | Kind.nestedSBC       => `(Pure.pure (DoResultSBC.«pureReturn» $val $u))
  | Kind.nestedPRBC      => `(Pure.pure (DoResultPRBC.«return» $val $u))

def continueToTerm : M Syntax := do
  let ctx ← read
  let u ← mkUVarTuple
  match ctx.kind with
  | Kind.regular         => unreachable!
  | Kind.forIn           => `(Pure.pure (ForInStep.yield $u))
  | Kind.forInWithReturn => `(Pure.pure (ForInStep.yield (MProd.mk none $u)))
  | Kind.nestedBC        => `(Pure.pure (DoResultBC.«continue» $u))
  | Kind.nestedPR        => unreachable!
  | Kind.nestedSBC       => `(Pure.pure (DoResultSBC.«continue» $u))
  | Kind.nestedPRBC      => `(Pure.pure (DoResultPRBC.«continue» $u))

def breakToTerm : M Syntax := do
  let ctx ← read
  let u ← mkUVarTuple
  match ctx.kind with
  | Kind.regular         => unreachable!
  | Kind.forIn           => `(Pure.pure (ForInStep.done $u))
```

```
  | Kind.forInWithReturn => `(Pure.pure (ForInStep.done (MProd.mk none $u)))
  | Kind.nestedBC        => `(Pure.pure (DoResultBC.«break» $u))
  | Kind.nestedPR        => unreachable!
  | Kind.nestedSBC       => `(Pure.pure (DoResultSBC.«break» $u))
  | Kind.nestedPRBC      => `(Pure.pure (DoResultPRBC.«break» $u))

def actionTerminalToTerm (action : Syntax) : M Syntax := withRef action <| withFreshMacroScope do
  let ctx ← read
  let u ← mkUVarTuple
  match ctx.kind with
  | Kind.regular         => if ctx.uvars.isEmpty then pure action else `(Bind.bind $action fun y => Pure.pure (MProd.mk y $u))
  | Kind.forIn           => `(Bind.bind $action fun (_ : PUnit) => Pure.pure (ForInStep.yield $u))
  | Kind.forInWithReturn => `(Bind.bind $action fun (_ : PUnit) => Pure.pure (ForInStep.yield (MProd.mk none $u)))
  | Kind.nestedBC        => unreachable!
  | Kind.nestedPR        => `(Bind.bind $action fun y => (Pure.pure (DoResultPR.«pure» y $u)))
  | Kind.nestedSBC       => `(Bind.bind $action fun y => (Pure.pure (DoResultSBC.«pureReturn» y $u)))
  | Kind.nestedPRBC      => `(Bind.bind $action fun y => (Pure.pure (DoResultPRBC.«pure» y $u)))

def seqToTerm (action : Syntax) (k : Syntax) : M Syntax := withRef action <| withFreshMacroScope do
  if action.getKind == `Lean.Parser.Term.doDbgTrace then
    let msg := action[1]
    `(dbg_trace $msg; $k)
  else if action.getKind == `Lean.Parser.Term.doAssert then
    let cond := action[1]
    `(assert! $cond; $k)
  else
    let action ← withRef action `(($action : $((←read).m) PUnit))
    `(Bind.bind $action (fun (_ : PUnit) => $k))

def declToTerm (decl : Syntax) (k : Syntax) : M Syntax := withRef decl <| withFreshMacroScope do
  let kind := decl.getKind
  if kind == `Lean.Parser.Term.doLet then
    let letDecl := decl[2]
    `(let $letDecl:letDecl; $k)
  else if kind == `Lean.Parser.Term.doLetRec then
    let letRecToken := decl[0]
    let letRecDecls := decl[1]
    pure $ mkNode `Lean.Parser.Term.letrec #[letRecToken, letRecDecls, mkNullNode, k]
  else if kind == `Lean.Parser.Term.doLetArrow then
    let arg := decl[2]
    let ref := arg
    if arg.getKind == `Lean.Parser.Term.doIdDecl then
      let id     := arg[0]
      let type   := expandOptType ref arg[1]
      let doElem := arg[3]
      -- `doElem` must be a `doExpr action`. See `doLetArrowToCode`
      match isDoExpr? doElem with
```

```
      | some action =>
        let action ← withRef action `(($action : $((← read).m) $type))
        `(Bind.bind $action (fun ($id:ident : $type) => $k))
      | none       => Macro.throwErrorAt decl "unexpected kind of 'do' declaration"
    else
      Macro.throwErrorAt decl "unexpected kind of 'do' declaration"
  else if kind == `Lean.Parser.Term.doHave then
    -- The `have` term is of the form  `"have " >> haveDecl >> optSemicolon termParser`
    let args := decl.getArgs
    let args := args ++ #[mkNullNode /- optional ';' -/, k]
    pure $ mkNode `Lean.Parser.Term.«have» args
  else
    Macro.throwErrorAt decl "unexpected kind of 'do' declaration"

def reassignToTerm (reassign : Syntax) (k : Syntax) : MacroM Syntax := withRef reassign <| withFreshMacroScope do
  let kind := reassign.getKind
  if kind == `Lean.Parser.Term.doReassign then
    -- doReassign := leading_parser (letIdDecl <|> letPatDecl)
    let arg := reassign[0]
    if arg.getKind == `Lean.Parser.Term.letIdDecl then
      -- letIdDecl := leading_parser ident >> many (ppSpace >> bracketedBinder) >> optType >>  " := " >> termParser
      let x   := arg[0]
      let val := arg[4]
      let newVal ← `(ensureTypeOf% $x $(quote "invalid reassignment, value") $val)
      let arg := arg.setArg 4 newVal
      let letDecl := mkNode `Lean.Parser.Term.letDecl #[arg]
      `(let $letDecl:letDecl; $k)
    else
      -- TODO: ensure the types did not change
      let letDecl := mkNode `Lean.Parser.Term.letDecl #[arg]
      `(let $letDecl:letDecl; $k)
  else
    -- Note that `doReassignArrow` is expanded by `doReassignArrowToCode
    Macro.throwErrorAt reassign "unexpected kind of 'do' reassignment"

def mkIte (optIdent : Syntax) (cond : Syntax) (thenBranch : Syntax) (elseBranch : Syntax) : MacroM Syntax := do
  if optIdent.isNone then
    `(ite $cond $thenBranch $elseBranch)
  else
    let h := optIdent[0]
    `(dite $cond (fun $h => $thenBranch) (fun $h => $elseBranch))

def mkJoinPoint (j : Name) (ps : Array (Name × Bool)) (body : Syntax) (k : Syntax) : M Syntax := withRef body <| withFreshMacroScope do
  let pTypes ← ps.mapM fun ⟨id, useTypeOf⟩ => do if useTypeOf then `(typeOf% $(← mkIdentFromRef id)) else `(_)
  let ps     ← ps.mapM fun ⟨id, useTypeOf⟩ => mkIdentFromRef id
  /-
  We use `let_delayed` instead of `let` for joinpoints to make sure `$k` is elaborated before `$body`.
```

```
  By elaborating `$k` first, we "learn" more about `$body`'s type.
  For example, consider the following example `do` expression
  ```
  def f (x : Nat) : IO Unit := do
  if x > 0 then
    IO.println "x is not zero" -- Error is here
  IO.mkRef true
  ```
  it is expanded into
  ```
  def f (x : Nat) : IO Unit := do
  let jp (u : Unit) : IO _ :=
    IO.mkRef true;
  if x > 0 then
    IO.println "not zero"
    jp ()
  else
    jp ()
  ```
  If we use the regular `let` instead of `let_delayed`, the joinpoint `jp` will be elaborated and its type will be inferred to be `Unit
  Then, we get a typing error at `jp ()`. By using `let_delayed`, we first elaborate `if x > 0 ...` and learn that `jp` has type `Unit →
  Then, we get the expected type mismatch error at `IO.mkRef true`. -/
  `(let_delayed $(← mkIdentFromRef j):ident $[($ps : $pTypes)]* : $((← read).m) _ := $body; $k)

def mkJmp (ref : Syntax) (j : Name) (args : Array Syntax) : Syntax :=
  Syntax.mkApp (mkIdentFrom ref j) args

partial def toTerm : Code → M Syntax
  | Code.«return» ref val   => withRef ref <| returnToTerm val
  | Code.«continue» ref     => withRef ref continueToTerm
  | Code.«break» ref        => withRef ref breakToTerm
  | Code.action e           => actionTerminalToTerm e
  | Code.joinpoint j ps b k => do mkJoinPoint j ps (← toTerm b) (← toTerm k)
  | Code.jmp ref j args     => pure $ mkJmp ref j args
  | Code.decl _ stx k       => do declToTerm stx (← toTerm k)
  | Code.reassign _ stx k   => do reassignToTerm stx (← toTerm k)
  | Code.seq stx k          => do seqToTerm stx (← toTerm k)
  | Code.ite ref _ o c t e  => withRef ref <| do mkIte o c (← toTerm t) (← toTerm e)
  | Code.«match» ref genParam discrs optType alts => do
    let mut termAlts := #[]
    for alt in alts do
      let rhs ← toTerm alt.rhs
      let termAlt := mkNode `Lean.Parser.Term.matchAlt #[mkAtomFrom alt.ref "|", alt.patterns, mkAtomFrom alt.ref "=>", rhs]
      termAlts := termAlts.push termAlt
    let termMatchAlts := mkNode `Lean.Parser.Term.matchAlts #[mkNullNode termAlts]
    pure $ mkNode `Lean.Parser.Term.«match» #[mkAtomFrom ref "match", genParam, discrs, optType, mkAtomFrom ref "with", termMatchAlts]
```

```
def run (code : Code) (m : Syntax) (uvars : Array Name := #[]) (kind := Kind.regular) : MacroM Syntax := do
  let term ← toTerm code { m := m, kind := kind, uvars := uvars }
  pure term

/- Given
   - `a` is true if the code block has a `Code.action _` exit point
   - `r` is true if the code block has a `Code.return _ _` exit point
   - `bc` is true if the code block has a `Code.break _` or `Code.continue _` exit point

   generate Kind. See comment at the beginning of the `ToTerm` namespace. -/
def mkNestedKind (a r bc : Bool) : Kind :=
  match a, r, bc with
  | true,  false, false => Kind.regular
  | false, true,  false => Kind.regular
  | false, false, true  => Kind.nestedBC
  | true,  true,  false => Kind.nestedPR
  | true,  false, true  => Kind.nestedSBC
  | false, true,  true  => Kind.nestedSBC
  | true,  true,  true  => Kind.nestedPRBC
  | false, false, false => unreachable!

def mkNestedTerm (code : Code) (m : Syntax) (uvars : Array Name) (a r bc : Bool) : MacroM Syntax := do
  ToTerm.run code m uvars (mkNestedKind a r bc)

/- Given a term `term` produced by `ToTerm.run`, pattern match on its result.
   See comment at the beginning of the `ToTerm` namespace.

   - `a` is true if the code block has a `Code.action _` exit point
   - `r` is true if the code block has a `Code.return _ _` exit point
   - `bc` is true if the code block has a `Code.break _` or `Code.continue _` exit point

   The result is a sequence of `doElem` -/
def matchNestedTermResult (term : Syntax) (uvars : Array Name) (a r bc : Bool) : MacroM (List Syntax) := do
  let toDoElems (auxDo : Syntax) : List Syntax := getDoSeqElems (getDoSeq auxDo)
  let u ← mkTuple (← uvars.mapM mkIdentFromRef)
  match a, r, bc with
  | true, false, false =>
    if uvars.isEmpty then
      toDoElems (← `(do $term:term))
    else
      toDoElems (← `(do let r ← $term:term; $u:term := r.2; pure r.1))
  | false, true, false =>
    if uvars.isEmpty then
      toDoElems (← `(do let r ← $term:term; return r))
    else
      toDoElems (← `(do let r ← $term:term; $u:term := r.2; return r.1))
  | false, false, true => toDoElems <$>
```

```
    `(do let r ← $term:term;
         match r with
         | DoResultBC.«break» u => $u:term := u; break
         | DoResultBC.«continue» u => $u:term := u; continue)
  | true, true, false => toDoElems <$>
    `(do let r ← $term:term;
         match r with
         | DoResultPR.«pure» a u => $u:term := u; pure a
         | DoResultPR.«return» b u => $u:term := u; return b)
  | true, false, true => toDoElems <$>
    `(do let r ← $term:term;
         match r with
         | DoResultSBC.«pureReturn» a u => $u:term := u; pure a
         | DoResultSBC.«break» u => $u:term := u; break
         | DoResultSBC.«continue» u => $u:term := u; continue)
  | false, true, true => toDoElems <$>
    `(do let r ← $term:term;
         match r with
         | DoResultSBC.«pureReturn» a u => $u:term := u; return a
         | DoResultSBC.«break» u => $u:term := u; break
         | DoResultSBC.«continue» u => $u:term := u; continue)
  | true, true, true => toDoElems <$>
    `(do let r ← $term:term;
         match r with
         | DoResultPRBC.«pure» a u => $u:term := u; pure a
         | DoResultPRBC.«return» a u => $u:term := u; return a
         | DoResultPRBC.«break» u => $u:term := u; break
         | DoResultPRBC.«continue» u => $u:term := u; continue)
  | false, false, false => unreachable!

end ToTerm

def isMutableLet (doElem : Syntax) : Bool :=
  let kind := doElem.getKind
  (kind == `Lean.Parser.Term.doLetArrow || kind == `Lean.Parser.Term.doLet)
  &&
  !doElem[1].isNone

namespace ToCodeBlock

structure Context where
  ref         : Syntax
  m           : Syntax -- Syntax representing the monad associated with the do notation.
  mutableVars : NameSet := {}
  insideFor   : Bool := false

abbrev M := ReaderT Context TermElabM
```

```
@[inline] def withNewMutableVars {α} (newVars : Array Name) (mutable : Bool) (x : M α) : M α :=
  withReader (fun ctx => if mutable then { ctx with mutableVars := insertVars ctx.mutableVars newVars } else ctx) x

def checkReassignable (xs : Array Name) : M Unit := do
  let throwInvalidReassignment (x : Name) : M Unit :=
    throwError "'{x.simpMacroScopes}' cannot be reassigned"
  let ctx ← read
  for x in xs do
    unless ctx.mutableVars.contains x do
      throwInvalidReassignment x

@[inline] def withFor {α} (x : M α) : M α :=
  withReader (fun ctx => { ctx with insideFor := true }) x

structure ToForInTermResult where
  uvars      : Array Name
  term       : Syntax

def mkForInBody  (x : Syntax) (forInBody : CodeBlock) : M ToForInTermResult := do
  let ctx ← read
  let uvars := forInBody.uvars
  let uvars := nameSetToArray uvars
  let term ← liftMacroM $ ToTerm.run forInBody.code ctx.m uvars (if hasReturn forInBody.code then ToTerm.Kind.forInWithReturn else ToTer
  pure ⟨uvars, term⟩

def ensureInsideFor : M Unit :=
  unless (← read).insideFor do
    throwError "invalid 'do' element, it must be inside 'for'"

def ensureEOS (doElems : List Syntax) : M Unit :=
  unless doElems.isEmpty do
    throwError "must be last element in a 'do' sequence"

private partial def expandLiftMethodAux (inQuot : Bool) (inBinder : Bool) : Syntax → StateT (List Syntax) MacroM Syntax
  | stx@(Syntax.node k args) =>
    if liftMethodDelimiter k then
      return stx
    else if k == `Lean.Parser.Term.liftMethod && !inQuot then withFreshMacroScope do
      if inBinder then
        Macro.throwErrorAt stx "cannot lift `(<- ...)` over a binder, this error usually happens when you are trying to lift a method ne
      let term := args[1]
      let term ← expandLiftMethodAux inQuot inBinder term
      let auxDoElem ← `(doElem| let a ← $term:term)
      modify fun s => s ++ [auxDoElem]
      `(a)
    else do
```

```lean
      let inAntiquot := stx.isAntiquot && !stx.isEscapedAntiquot
      let inBinder   := inBinder || (!inQuot && liftMethodForbiddenBinder k)
      let args ← args.mapM (expandLiftMethodAux (inQuot && !inAntiquot || stx.isQuot) inBinder)
      return Syntax.node k args
  | stx => pure stx

def expandLiftMethod (doElem : Syntax) : MacroM (List Syntax × Syntax) := do
  if !hasLiftMethod doElem then
    pure ([], doElem)
  else
    let (doElem, doElemsNew) ← (expandLiftMethodAux false false doElem).run []
    pure (doElemsNew, doElem)

def checkLetArrowRHS (doElem : Syntax) : M Unit := do
  let kind := doElem.getKind
  if kind == `Lean.Parser.Term.doLetArrow ||
     kind == `Lean.Parser.Term.doLet ||
     kind == `Lean.Parser.Term.doLetRec ||
     kind == `Lean.Parser.Term.doHave ||
     kind == `Lean.Parser.Term.doReassign ||
     kind == `Lean.Parser.Term.doReassignArrow then
    throwErrorAt doElem "invalid kind of value '{kind}' in an assignment"

/- Generate `CodeBlock` for `doReturn` which is of the form
   ```
   "return " >> optional termParser
   ```
   `doElems` is only used for sanity checking. -/
def doReturnToCode (doReturn : Syntax) (doElems: List Syntax) : M CodeBlock := withRef doReturn do
  ensureEOS doElems
  let argOpt := doReturn[1]
  let arg ← if argOpt.isNone then liftMacroM mkUnit else pure argOpt[0]
  return mkReturn (← getRef) arg

structure Catch where
  x         : Syntax
  optType   : Syntax
  codeBlock : CodeBlock

def getTryCatchUpdatedVars (tryCode : CodeBlock) (catches : Array Catch) (finallyCode? : Option CodeBlock) : NameSet :=
  let ws := tryCode.uvars
  let ws := catches.foldl (fun ws alt => union alt.codeBlock.uvars ws) ws
  let ws := match finallyCode? with
    | none   => ws
    | some c => union c.uvars ws
  ws
```

```
def tryCatchPred (tryCode : CodeBlock) (catches : Array Catch) (finallyCode? : Option CodeBlock) (p : Code → Bool) : Bool :=
  p tryCode.code ||
  catches.any (fun «catch» => p «catch».codeBlock.code) ||
  match finallyCode? with
  | none => false
  | some finallyCode => p finallyCode.code

mutual
  /- "Concatenate" `c` with `doSeqToCode doElems` -/
  partial def concatWith (c : CodeBlock) (doElems : List Syntax) : M CodeBlock :=
    match doElems with
    | [] => pure c
    | nextDoElem :: _  => do
      let k ← doSeqToCode doElems
      let ref := nextDoElem
      concat c ref none k

  /- Generate `CodeBlock` for `doLetArrow; doElems`
     `doLetArrow` is of the form
     ```
     "let " >> optional "mut " >> (doIdDecl <|> doPatDecl)
     ```
     where
     ```
     def doIdDecl   := leading_parser ident >> optType >> leftArrow >> doElemParser
     def doPatDecl := leading_parser termParser >> leftArrow >> doElemParser >> optional (" | " >> doElemParser)
     ```
  -/
  partial def doLetArrowToCode (doLetArrow : Syntax) (doElems : List Syntax) : M CodeBlock := do
    let ref     := doLetArrow
    let decl    := doLetArrow[2]
    if decl.getKind == `Lean.Parser.Term.doIdDecl then
      let y := decl[0].getId
      let doElem := decl[3]
      let k ← withNewMutableVars #[y] (isMutableLet doLetArrow) (doSeqToCode doElems)
      match isDoExpr? doElem with
      | some action => pure $ mkVarDeclCore #[y] doLetArrow k
      | none =>
        checkLetArrowRHS doElem
        let c ← doSeqToCode [doElem]
        match doElems with
        | []       => pure c
        | kRef::_  => concat c kRef y k
    else if decl.getKind == `Lean.Parser.Term.doPatDecl then
      let pattern := decl[0]
      let doElem  := decl[2]
      let optElse := decl[3]
```

```
    if optElse.isNone then withFreshMacroScope do
      let auxDo ←
        if isMutableLet doLetArrow then
          `(do let discr ← $doElem; let mut $pattern:term := discr)
        else
          `(do let discr ← $doElem; let $pattern:term := discr)
      doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
    else
      if isMutableLet doLetArrow then
        throwError "'mut' is currently not supported in let-decls with 'else' case"
      let contSeq := mkDoSeq doElems.toArray
      let elseSeq := mkSingletonDoSeq optElse[1]
      let auxDo ← `(do let discr ← $doElem; match discr with | $pattern:term => $contSeq | _ => $elseSeq)
      doSeqToCode <| getDoSeqElems (getDoSeq auxDo)
  else
    throwError "unexpected kind of 'do' declaration"


/- Generate `CodeBlock` for `doReassignArrow; doElems`
   `doReassignArrow` is of the form
   ```
   (doIdDecl <|> doPatDecl)
   ```
-/
partial def doReassignArrowToCode (doReassignArrow : Syntax) (doElems : List Syntax) : M CodeBlock := do
  let ref  := doReassignArrow
  let decl := doReassignArrow[0]
  if decl.getKind == `Lean.Parser.Term.doIdDecl then
    let doElem := decl[3]
    let y      := decl[0]
    let auxDo ← `(do let r ← $doElem; $y:ident := r)
    doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
  else if decl.getKind == `Lean.Parser.Term.doPatDecl then
    let pattern := decl[0]
    let doElem  := decl[2]
    let optElse := decl[3]
    if optElse.isNone then withFreshMacroScope do
      let auxDo ← `(do let discr ← $doElem; $pattern:term := discr)
      doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
    else
      throwError "reassignment with `|` (i.e., \"else clause\") is not currently supported"
  else
    throwError "unexpected kind of 'do' reassignment"

/- Generate `CodeBlock` for `doIf; doElems`
   `doIf` is of the form
   ```
```
```

```
  "if " >> optIdent >> termParser >> " then " >> doSeq
   >> many (group (try (group (" else " >> " if ")) >> optIdent >> termParser >> " then " >> doSeq))
   >> optional (" else " >> doSeq)
  ```  -/
partial def doIfToCode (doIf : Syntax) (doElems : List Syntax) : M CodeBlock := do
  let view ← liftMacroM $ mkDoIfView doIf
  let thenBranch ← doSeqToCode (getDoSeqElems view.thenBranch)
  let elseBranch ← doSeqToCode (getDoSeqElems view.elseBranch)
  let ite ← mkIte view.ref view.optIdent view.cond thenBranch elseBranch
  concatWith ite doElems

/- Generate `CodeBlock` for `doUnless; doElems`
  `doUnless` is of the form
  ```
  "unless " >> termParser >> "do " >> doSeq
  ```  -/
partial def doUnlessToCode (doUnless : Syntax) (doElems : List Syntax) : M CodeBlock := withRef doUnless do
  let ref   := doUnless
  let cond := doUnless[1]
  let doSeq := doUnless[3]
  let body ← doSeqToCode (getDoSeqElems doSeq)
  let unlessCode ← liftMacroM <| mkUnless cond body
  concatWith unlessCode doElems

/- Generate `CodeBlock` for `doFor; doElems`
  `doFor` is of the form
  ```
  def doForDecl := leading_parser termParser >> " in " >> withForbidden "do" termParser
  def doFor := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq
  ```
-/
partial def doForToCode (doFor : Syntax) (doElems : List Syntax) : M CodeBlock := do
  let doForDecls := doFor[1].getSepArgs
  if doForDecls.size > 1 then
    /-
      Expand
      ```
      for x in xs, y in ys do
        body
      ```
      into
      ```
      let s := toStream ys
      for x in xs do
        match Stream.next? s with
        | none => break
        | some (y, s') =>
```

```
            s := s'
            body
        ```
      -/
      -- Extract second element
      let doForDecl := doForDecls[1]
      let y := doForDecl[0]
      let ys := doForDecl[2]
      let doForDecls := doForDecls.eraseIdx 1
      let body := doFor[3]
      withFreshMacroScope do
        let toStreamFn ← withRef ys `(toStream)
        let auxDo ←
          `(do let mut s := $toStreamFn:ident $ys
               for $doForDecls:doForDecl,* do
                 match Stream.next? s with
                 | none => break
                 | some ($y, s') =>
                   s := s'
                   do $body)
        doSeqToCode (getDoSeqElems (getDoSeq auxDo) ++ doElems)
    else withRef doFor do
      let x   := doForDecls[0][0]
      let xs   := doForDecls[0][2]
      let forElems := getDoSeqElems doFor[3]
      let forInBodyCodeBlock ← withFor (doSeqToCode forElems)
      let ⟨uvars, forInBody⟩ ← mkForInBody x forInBodyCodeBlock
      let uvarsTuple ← liftMacroM do mkTuple (← uvars.mapM mkIdentFromRef)
      if hasReturn forInBodyCodeBlock.code then
        let forInBody ← liftMacroM <| destructTuple uvars (← `(r)) forInBody
        let forInTerm ← `(forIn% $(xs) (MProd.mk none $uvarsTuple) fun $x r => let r := r.2; $forInBody)
        let auxDo ← `(do let r ← $forInTerm:term;
                         $uvarsTuple:term := r.2;
                         match r.1 with
                         | none => Pure.pure (ensureExpectedType% "type mismatch, 'for'" PUnit.unit)
                         | some a => return ensureExpectedType% "type mismatch, 'for'" a)
        doSeqToCode (getDoSeqElems (getDoSeq auxDo) ++ doElems)
      else
        let forInBody ← liftMacroM <| destructTuple uvars (← `(r)) forInBody
        let forInTerm ← `(forIn% $(xs) $uvarsTuple fun $x r => $forInBody)
        if doElems.isEmpty then
          let auxDo ← `(do let r ← $forInTerm:term;
                           $uvarsTuple:term := r;
                           Pure.pure (ensureExpectedType% "type mismatch, 'for'" PUnit.unit))
          doSeqToCode <| getDoSeqElems (getDoSeq auxDo)
        else
          let auxDo ← `(do let r ← $forInTerm:term; $uvarsTuple:term := r)
```
```

```
        doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems

/-- Generate `CodeBlock` for `doMatch; doElems` -/
partial def doMatchToCode (doMatch : Syntax) (doElems: List Syntax) : M CodeBlock := do
  let ref       := doMatch
  let genParam  := doMatch[1]
  let discrs    := doMatch[2]
  let optType   := doMatch[3]
  let matchAlts := doMatch[5][0].getArgs -- Array of `doMatchAlt`
  let alts ←  matchAlts.mapM fun matchAlt => do
    let patterns := matchAlt[1]
    let vars ← getPatternsVarsEx patterns.getSepArgs
    let rhs  := matchAlt[3]
    let rhs ← doSeqToCode (getDoSeqElems rhs)
    pure { ref := matchAlt, vars := vars, patterns := patterns, rhs := rhs : Alt CodeBlock }
  let matchCode ← mkMatch ref genParam discrs optType alts
  concatWith matchCode doElems

/--
  Generate `CodeBlock` for `doTry; doElems`
  ```
  def doTry := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally
  def doCatch   := leading_parser "catch " >> binderIdent >> optional (":" >> termParser) >> darrow >> doSeq
  def doCatchMatch := leading_parser "catch " >> doMatchAlts
  def doFinally   := leading_parser "finally " >> doSeq
  ```
-/
partial def doTryToCode (doTry : Syntax) (doElems: List Syntax) : M CodeBlock := do
  let ref := doTry
  let tryCode ← doSeqToCode (getDoSeqElems doTry[1])
  let optFinally := doTry[3]
  let catches ← doTry[2].getArgs.mapM fun catchStx => do
    if catchStx.getKind == `Lean.Parser.Term.doCatch then
      let x       := catchStx[1]
      let optType := catchStx[2]
      let c ← doSeqToCode (getDoSeqElems catchStx[4])
      pure { x := x, optType := optType, codeBlock := c : Catch }
    else if catchStx.getKind == `Lean.Parser.Term.doCatchMatch then
      let matchAlts := catchStx[1]
      let x ← `(ex)
      let auxDo ← `(do match ex with $matchAlts)
      let c ← doSeqToCode (getDoSeqElems (getDoSeq auxDo))
      pure { x := x, codeBlock := c, optType := mkNullNode : Catch }
    else
      throwError "unexpected kind of 'catch'"
  let finallyCode? ← if optFinally.isNone then pure none else some <$> doSeqToCode (getDoSeqElems optFinally[0][1])
  if catches.isEmpty && finallyCode?.isNone then
```

```
    throwError "invalid 'try', it must have a 'catch' or 'finally'"
  let ctx ← read
  let ws    := getTryCatchUpdatedVars tryCode catches finallyCode?
  let uvars := nameSetToArray ws
  let a     := tryCatchPred tryCode catches finallyCode? hasTerminalAction
  let r     := tryCatchPred tryCode catches finallyCode? hasReturn
  let bc    := tryCatchPred tryCode catches finallyCode? hasBreakContinue
  let toTerm (codeBlock : CodeBlock) : M Syntax := do
    let codeBlock ← liftM $ extendUpdatedVars codeBlock ws
    liftMacroM $ ToTerm.mkNestedTerm codeBlock.code ctx.m uvars a r bc
  let term ← toTerm tryCode
  let term ← catches.foldlM
    (fun term «catch» => do
      let catchTerm ← toTerm «catch».codeBlock
      if catch.optType.isNone then
        `(MonadExcept.tryCatch $term (fun $(«catch».x):ident => $catchTerm))
      else
        let type := «catch».optType[1]
        `(tryCatchThe $type $term (fun $(«catch».x):ident => $catchTerm)))
    term
  let term ← match finallyCode? with
    | none             => pure term
    | some finallyCode => withRef optFinally do
      unless finallyCode.uvars.isEmpty do
        throwError "'finally' currently does not support reassignments"
      if hasBreakContinueReturn finallyCode.code then
        throwError "'finally' currently does 'return', 'break', nor 'continue'"
      let finallyTerm ← liftMacroM <| ToTerm.run finallyCode.code ctx.m {} ToTerm.Kind.regular
      `(tryFinally $term $finallyTerm)
  let doElemsNew ← liftMacroM <| ToTerm.matchNestedTermResult term uvars a r bc
  doSeqToCode (doElemsNew ++ doElems)

partial def doSeqToCode : List Syntax → M CodeBlock
  | [] => do liftMacroM mkPureUnitAction
  | doElem::doElems => withIncRecDepth <| withRef doElem do
    checkMaxHeartbeats "'do'-expander"
    match (← liftMacroM <| expandMacro? doElem) with
    | some doElem => doSeqToCode (doElem::doElems)
    | none =>
    match (← liftMacroM <| expandDoIf? doElem) with
    | some doElem => doSeqToCode (doElem::doElems)
    | none =>
      let (liftedDoElems, doElem) ← liftM (liftMacroM <| expandLiftMethod doElem : TermElabM _)
      if !liftedDoElems.isEmpty then
        doSeqToCode (liftedDoElems ++ [doElem] ++ doElems)
      else
        let ref := doElem
```

```
        let concatWithRest (c : CodeBlock) : M CodeBlock := concatWith c doElems
        let k := doElem.getKind
        if k == `Lean.Parser.Term.doLet then
          let vars ← getDoLetVars doElem
          mkVarDeclCore vars doElem <$> withNewMutableVars vars (isMutableLet doElem) (doSeqToCode doElems)
        else if k == `Lean.Parser.Term.doHave then
          let var := getDoHaveVar doElem
          mkVarDeclCore #[var] doElem <$> (doSeqToCode doElems)
        else if k == `Lean.Parser.Term.doLetRec then
          let vars ← getDoLetRecVars doElem
          mkVarDeclCore vars doElem <$> (doSeqToCode doElems)
        else if k == `Lean.Parser.Term.doReassign then
          let vars ← getDoReassignVars doElem
          checkReassignable vars
          let k ← doSeqToCode doElems
          mkReassignCore vars doElem k
        else if k == `Lean.Parser.Term.doLetArrow then
          doLetArrowToCode doElem doElems
        else if k == `Lean.Parser.Term.doReassignArrow then
          doReassignArrowToCode doElem doElems
        else if k == `Lean.Parser.Term.doIf then
          doIfToCode doElem doElems
        else if k == `Lean.Parser.Term.doUnless then
          doUnlessToCode doElem doElems
        else if k == `Lean.Parser.Term.doFor then withFreshMacroScope do
          doForToCode doElem doElems
        else if k == `Lean.Parser.Term.doMatch then
          doMatchToCode doElem doElems
        else if k == `Lean.Parser.Term.doTry then
          doTryToCode doElem doElems
        else if k == `Lean.Parser.Term.doBreak then
          ensureInsideFor
          ensureEOS doElems
          return mkBreak ref
        else if k == `Lean.Parser.Term.doContinue then
          ensureInsideFor
          ensureEOS doElems
          return mkContinue ref
        else if k == `Lean.Parser.Term.doReturn then
          doReturnToCode doElem doElems
        else if k == `Lean.Parser.Term.doDbgTrace then
          return mkSeq doElem (← doSeqToCode doElems)
        else if k == `Lean.Parser.Term.doAssert then
          return mkSeq doElem (← doSeqToCode doElems)
        else if k == `Lean.Parser.Term.doNested then
          let nestedDoSeq := doElem[1]
          doSeqToCode (getDoSeqElems nestedDoSeq ++ doElems)
```

```
        else if k == `Lean.Parser.Term.doExpr then
          let term := doElem[0]
          if doElems.isEmpty then
            return mkTerminalAction term
          else
            return mkSeq term (← doSeqToCode doElems)
        else
          throwError "unexpected do-element\n{doElem}"
end

def run (doStx : Syntax) (m : Syntax) : TermElabM CodeBlock :=
  (doSeqToCode <| getDoSeqElems <| getDoSeq doStx).run { ref := doStx, m := m }

end ToCodeBlock

/- Create a synthetic metavariable `?m` and assign `m` to it.
   We use `?m` to refer to `m` when expanding the `do` notation. -/
private def mkMonadAlias (m : Expr) : TermElabM Syntax := do
  let result ← `(?m)
  let mType ← inferType m
  let mvar ← elabTerm result mType
  assignExprMVar mvar.mvarId! m
  pure result

@[builtinTermElab «do»]
def elabDo : TermElab := fun stx expectedType? => do
  tryPostponeIfNoneOrMVar expectedType?
  let bindInfo ← extractBind expectedType?
  let m ← mkMonadAlias bindInfo.m
  let codeBlock ← ToCodeBlock.run stx m
  let stxNew ← liftMacroM $ ToTerm.run codeBlock.code m
  trace[Elab.do] stxNew
  withMacroExpansion stx stxNew $ elabTermEnsuringType stxNew bindInfo.expectedType

end Do

builtin_initialize registerTraceClass `Elab.do

private def toDoElem (newKind : SyntaxNodeKind) : Macro := fun stx => do
  let stx := stx.setKind newKind
  withRef stx `(do $stx:doElem)

@[builtinMacro Lean.Parser.Term.termFor]
def expandTermFor : Macro := toDoElem `Lean.Parser.Term.doFor

@[builtinMacro Lean.Parser.Term.termTry]
def expandTermTry : Macro := toDoElem `Lean.Parser.Term.doTry
```

```
@[builtinMacro Lean.Parser.Term.termUnless]
def expandTermUnless : Macro := toDoElem `Lean.Parser.Term.doUnless

@[builtinMacro Lean.Parser.Term.termReturn]
def expandTermReturn : Macro := toDoElem `Lean.Parser.Term.doReturn

end Lean.Elab.Term
::::::::::::::
Elab/Exception.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.InternalExceptionId
import Lean.Meta.Basic

namespace Lean.Elab

builtin_initialize postponeExceptionId : InternalExceptionId ← registerInternalExceptionId `postpone
builtin_initialize unsupportedSyntaxExceptionId : InternalExceptionId ← registerInternalExceptionId `unsupportedSyntax
builtin_initialize abortCommandExceptionId : InternalExceptionId ← registerInternalExceptionId `abortCommandElab
builtin_initialize abortTermExceptionId : InternalExceptionId ← registerInternalExceptionId `abortTermElab
builtin_initialize autoBoundImplicitExceptionId : InternalExceptionId ← registerInternalExceptionId `autoBoundImplicit

def throwPostpone [MonadExceptOf Exception m] : m α :=
  throw $ Exception.internal postponeExceptionId

def throwUnsupportedSyntax [MonadExceptOf Exception m] : m α :=
  throw $ Exception.internal unsupportedSyntaxExceptionId

def throwIllFormedSyntax [Monad m] [MonadError m] : m α :=
  throwError "ill-formed syntax"

def throwAutoBoundImplicitLocal [MonadExceptOf Exception m] (n : Name) : m α :=
  throw $ Exception.internal autoBoundImplicitExceptionId <| KVMap.empty.insert `localId n

def isAutoBoundImplicitLocalException? (ex : Exception) : Option Name :=
  match ex with
  | Exception.internal id k =>
    if id == autoBoundImplicitExceptionId then
      some <| k.getName `localId `x
    else
      none
  | _ => none
```

```
def throwAlreadyDeclaredUniverseLevel [Monad m] [MonadError m] (u : Name) : m α :=
  throwError "a universe level named '{u}' has already been declared"

-- Throw exception to abort elaboration of the current command without producing any error message
def throwAbortCommand {α m} [MonadExcept Exception m] : m α :=
  throw <| Exception.internal abortCommandExceptionId

-- Throw exception to abort elaboration of the current term without producing any error message
def throwAbortTerm {α m} [MonadExcept Exception m] : m α :=
  throw <| Exception.internal abortTermExceptionId

def isAbortExceptionId (id : InternalExceptionId) : Bool :=
  id == abortCommandExceptionId || id == abortTermExceptionId

def mkMessageCore (fileName : String) (fileMap : FileMap) (msgData : MessageData) (severity : MessageSeverity) (pos : String.Pos) : Mess
  let pos := fileMap.toPosition pos
  { fileName := fileName, pos := pos, data := msgData, severity := severity }

end Lean.Elab
::::::::::::::
Elab/Extra.lean
::::::::::::::
/-
Copyright (c) 2021 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.App

/-
Auxiliary elaboration functions: AKA custom elaborators
-/

namespace Lean.Elab.Term
open Meta

@[builtinTermElab binrel] def elabBinRel : TermElab :=  fun stx expectedType? => do
  match (← resolveId? stx[1]) with
  | some f =>
    let s ← saveState
    let (lhs, rhs) ← withSynthesize (mayPostpone := true) do
      let mut lhs ← elabTerm stx[2] none
      let mut rhs ← elabTerm stx[3] none
      if lhs.isAppOfArity `OfNat.ofNat 3 then
        lhs ← ensureHasType (← inferType rhs) lhs
      else if rhs.isAppOfArity `OfNat.ofNat 3 then
```

```
        rhs ← ensureHasType (← inferType lhs) rhs
      return (lhs, rhs)
    let lhsType ← inferType lhs
    let rhsType ← inferType rhs
    let (lhs, rhs) ←
      try
        pure (lhs, ← withRef stx[3] do ensureHasType lhsType rhs)
      catch _ =>
        try
          pure (← withRef stx[2] do ensureHasType rhsType lhs, rhs)
        catch _ =>
          s.restore
          -- Use default approach
          let lhs ← elabTerm stx[2] none
          let rhs ← elabTerm stx[3] none
          let lhsType ← inferType lhs
          let rhsType ← inferType rhs
          pure (lhs, ← withRef stx[3] do ensureHasType lhsType rhs)
    elabAppArgs f #[] #[Arg.expr lhs, Arg.expr rhs] expectedType? (explicit := false) (ellipsis := false)
  | none   => throwUnknownConstant stx[1].getId

@[builtinTermElab forInMacro] def elabForIn : TermElab :=  fun stx expectedType? => do
  match stx with
  | `(forIn% $col $init $body) =>
      match (← isLocalIdent? col) with
      | none   => elabTerm (← `(let col := $col; forIn% col $init $body)) expectedType?
      | some colFVar =>
        tryPostponeIfNoneOrMVar expectedType?
        let m ← getMonad expectedType?
        let colType ← inferType colFVar
        let elemType ← mkFreshExprMVar (mkSort (mkLevelSucc (← mkFreshLevelMVar)))
        let forInInstance ←
          try
            mkAppM `ForIn #[m, colType, elemType]
          catch
            ex => tryPostpone; throwError "failed to construct 'ForIn' instance for collection{indentExpr colType}\nand monad{indentExpr
        match (← trySynthInstance forInInstance) with
        | LOption.some val =>
          let ref ← getRef
          let forInFn ← mkConst ``forIn
          let namedArgs : Array NamedArg := #[
            { ref := ref, name := `m, val := Arg.expr m},
            { ref := ref, name := `ρ, val := Arg.expr colType},
            { ref := ref, name := `α, val := Arg.expr elemType},
            { ref := ref, name := `self, val := Arg.expr forInInstance},
            { ref := ref, name := `inst, val := Arg.expr val} ]
          elabAppArgs forInFn #[] #[Arg.stx col, Arg.stx init, Arg.stx body] expectedType? (explicit := false) (ellipsis := false)
```

```
        | LOption.undef    => tryPostpone; throwFailure forInInstance
        | LOption.none     => throwFailure forInInstance
  | _ => throwUnsupportedSyntax
where
  getMonad (expectedType? : Option Expr) : TermElabM Expr := do
    match expectedType? with
    | none => throwError "invalid 'forIn%' notation, expected type is not available"
    | some expectedType =>
      match (← isTypeApp? expectedType) with
      | some (m, _) => return m
      | none => throwError "invalid 'forIn%' notation, expected type is not of of the form `M α`{indentExpr expectedType}"
  throwFailure (forInInstance : Expr) : TermElabM Expr :=
    throwError "failed to synthesize instance for 'forIn%' notation{indentExpr forInInstance}"

end Lean.Elab.Term
::::::::::::::
Elab/Frontend.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Elab.Import
import Lean.Elab.Command
import Lean.Util.Profile

namespace Lean.Elab.Frontend

structure State where
  commandState : Command.State
  parserState  : Parser.ModuleParserState
  cmdPos       : String.Pos
  commands     : Array Syntax := #[]

structure Context where
  inputCtx : Parser.InputContext

abbrev FrontendM := ReaderT Context $ StateRefT State IO

def setCommandState (commandState : Command.State) : FrontendM Unit :=
  modify fun s => { s with commandState := commandState }

@[inline] def runCommandElabM (x : Command.CommandElabM α) : FrontendM α := do
  let ctx ← read
  let s ← get
  let cmdCtx : Command.Context := { cmdPos := s.cmdPos, fileName := ctx.inputCtx.fileName, fileMap := ctx.inputCtx.fileMap }
```

```
  match ← liftM <| EIO.toIO' <| (x cmdCtx).run s.commandState with
  | Except.error e      => throw <| IO.Error.userError s!"unexpected internal error: {← e.toMessageData.toString}"
  | Except.ok (a, sNew) => setCommandState sNew; return a

def elabCommandAtFrontend (stx : Syntax) : FrontendM Unit := do
  runCommandElabM do
    let infoTreeEnabled := (← getInfoState).enabled
    if checkTraceOption (← getOptions) `Elab.info then
      enableInfoTree
    Command.elabCommand stx
    enableInfoTree infoTreeEnabled

def updateCmdPos : FrontendM Unit := do
  modify fun s => { s with cmdPos := s.parserState.pos }

def getParserState : FrontendM Parser.ModuleParserState := do pure (← get).parserState
def getCommandState : FrontendM Command.State := do pure (← get).commandState
def setParserState (ps : Parser.ModuleParserState) : FrontendM Unit := modify fun s => { s with parserState := ps }
def setMessages (msgs : MessageLog) : FrontendM Unit := modify fun s => { s with commandState := { s.commandState with messages := msgs
def getInputContext : FrontendM Parser.InputContext := do pure (← read).inputCtx

def processCommand : FrontendM Bool := do
  updateCmdPos
  let cmdState ← getCommandState
  let ictx ← getInputContext
  let pstate ← getParserState
  let scope := cmdState.scopes.head!
  let pmctx := { env := cmdState.env, options := scope.opts, currNamespace := scope.currNamespace, openDecls := scope.openDecls }
  let pos := ictx.fileMap.toPosition pstate.pos
  match profileit "parsing" scope.opts fun _ => Parser.parseCommand ictx pmctx pstate cmdState.messages with
  | (cmd, ps, messages) =>
    modify fun s => { s with commands := s.commands.push cmd }
    setParserState ps
    setMessages messages
    if Parser.isEOI cmd || Parser.isExitCommand cmd then
      pure true -- Done
    else
      profileitM IO.Error "elaboration" scope.opts <| elabCommandAtFrontend cmd
      pure false

partial def processCommands : FrontendM Unit := do
  let done ← processCommand
  unless done do
    processCommands

end Frontend
```

```
open Frontend

def IO.processCommands (inputCtx : Parser.InputContext) (parserState : Parser.ModuleParserState) (commandState : Command.State) : IO Sta
  let (_, s) ← (Frontend.processCommands.run { inputCtx := inputCtx }).run { commandState := commandState, parserState := parserState, c
  pure s

def process (input : String) (env : Environment) (opts : Options) (fileName : Option String := none) : IO (Environment × MessageLog) := 
  let fileName   := fileName.getD "<input>"
  let inputCtx   := Parser.mkInputContext input fileName
  let s ← IO.processCommands inputCtx { : Parser.ModuleParserState } (Command.mkState env {} opts)
  pure (s.commandState.env, s.commandState.messages)

builtin_initialize
  registerOption `printMessageEndPos { defValue := false, descr := "print end position of each message in addition to start position" }
  registerTraceClass `Elab.info

def getPrintMessageEndPos (opts : Options) : Bool :=
  opts.getBool `printMessageEndPos false

@[export lean_run_frontend]
def runFrontend (input : String) (opts : Options) (fileName : String) (mainModuleName : Name) : IO (Environment × Bool) := do
  let inputCtx := Parser.mkInputContext input fileName
  let (header, parserState, messages) ← Parser.parseHeader inputCtx
  let (env, messages) ← processHeader header opts messages inputCtx
  let env := env.setMainModule mainModuleName
  let s ← IO.processCommands inputCtx parserState (Command.mkState env messages opts)
  for msg in s.commandState.messages.toList do
    IO.print (← msg.toString (includeEndPos := getPrintMessageEndPos opts))
  pure (s.commandState.env, !s.commandState.messages.hasErrors)

end Lean.Elab
::::::::::::::
Elab/Import.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Module
namespace Lean.Elab

def headerToImports (header : Syntax) : List Import :=
  let imports := if header[0].isNone then [{ module := `Init : Import }] else []
  imports ++ header[1].getArgs.toList.map fun stx =>
    -- `stx` is of the form `(Module.import "import" "runtime"? id)
    let runtime := !stx[1].isNone
```

```
    let id      := stx[2].getId
    { module := id, runtimeOnly := runtime }

def processHeader (header : Syntax) (opts : Options) (messages : MessageLog) (inputCtx : Parser.InputContext) (trustLevel : UInt32 := 0)
    : IO (Environment × MessageLog) := do
  try
    let env ← importModules (headerToImports header) opts trustLevel
    pure (env, messages)
  catch e =>
    let env ← mkEmptyEnvironment
    let spos := header.getPos?.getD 0
    let pos  := inputCtx.fileMap.toPosition spos
    pure (env, messages.add { fileName := inputCtx.fileName, data := toString e, pos := pos })

def parseImports (input : String) (fileName : Option String := none) : IO (List Import × Position × MessageLog) := do
  let fileName := fileName.getD "<input>"
  let inputCtx := Parser.mkInputContext input fileName
  let (header, parserState, messages) ← Parser.parseHeader inputCtx
  pure (headerToImports header, inputCtx.fileMap.toPosition parserState.pos, messages)

@[export lean_print_imports]
def printImports (input : String) (fileName : Option String) : IO Unit := do
  let (deps, pos, log) ← parseImports input fileName
  for dep in deps do
    let fname ← findOLean dep.module
    IO.println fname

end Lean.Elab
::::::::::::::
Elab/Inductive.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.ReplaceLevel
import Lean.Util.ReplaceExpr
import Lean.Util.CollectLevelParams
import Lean.Util.Constructions
import Lean.Meta.CollectFVars
import Lean.Meta.SizeOf
import Lean.Elab.Command
import Lean.Elab.DefView
import Lean.Elab.DeclUtil
import Lean.Elab.Deriving.Basic

```

```
namespace Lean.Elab.Command
open Meta

builtin_initialize
  registerTraceClass `Elab.inductive

def checkValidInductiveModifier [Monad m] [MonadError m] (modifiers : Modifiers) : m Unit := do
  if modifiers.isNoncomputable then
    throwError "invalid use of 'noncomputable' in inductive declaration"
  if modifiers.isPartial then
    throwError "invalid use of 'partial' in inductive declaration"
  unless modifiers.attrs.size == 0 || (modifiers.attrs.size == 1 && modifiers.attrs[0].name == `class) do
    throwError "invalid use of attributes in inductive declaration"

def checkValidCtorModifier [Monad m] [MonadError m] (modifiers : Modifiers) : m Unit := do
  if modifiers.isNoncomputable then
    throwError "invalid use of 'noncomputable' in constructor declaration"
  if modifiers.isPartial then
    throwError "invalid use of 'partial' in constructor declaration"
  if modifiers.isUnsafe then
    throwError "invalid use of 'unsafe' in constructor declaration"
  if modifiers.attrs.size != 0 then
    throwError "invalid use of attributes in constructor declaration"

structure CtorView where
  ref       : Syntax
  modifiers : Modifiers
  inferMod  : Bool  -- true if `{}` is used in the constructor declaration
  declName  : Name
  binders   : Syntax
  type?     : Option Syntax
  deriving Inhabited

structure InductiveView where
  ref             : Syntax
  modifiers       : Modifiers
  shortDeclName   : Name
  declName        : Name
  levelNames      : List Name
  binders         : Syntax
  type?           : Option Syntax
  ctors           : Array CtorView
  derivingClasses : Array DerivingClassView
  deriving Inhabited

structure ElabHeaderResult where
  view       : InductiveView
```

```
  lctx       : LocalContext
  localInsts : LocalInstances
  params     : Array Expr
  type       : Expr
  deriving Inhabited

private partial def elabHeaderAux (views : Array InductiveView) (i : Nat) (acc : Array ElabHeaderResult) : TermElabM (Array ElabHeaderRe
  if h : i < views.size then
    let view := views.get ⟨i, h⟩
    let acc ← Term.withAutoBoundImplicit <| Term.elabBinders view.binders.getArgs fun params => do
      match view.type? with
      | none         =>
        let u ← mkFreshLevelMVar
        let type := mkSort u
        Term.synthesizeSyntheticMVarsNoPostponing
        let params ← Term.addAutoBoundImplicits params
        pure <| acc.push { lctx := (← getLCtx), localInsts := (← getLocalInstances), params := params, type := type, view := view }
      | some typeStx =>
        let type ← Term.elabType typeStx
        unless (← isTypeFormerType type) do
          throwErrorAt typeStx "invalid inductive type, resultant type is not a sort"
        Term.synthesizeSyntheticMVarsNoPostponing
        let params ← Term.addAutoBoundImplicits params
        pure <| acc.push { lctx := (← getLCtx), localInsts := (← getLocalInstances), params := params, type := type, view := view }
    elabHeaderAux views (i+1) acc
  else
    pure acc

private def checkNumParams (rs : Array ElabHeaderResult) : TermElabM Nat := do
  let numParams := rs[0].params.size
  for r in rs do
    unless r.params.size == numParams do
      throwErrorAt r.view.ref "invalid inductive type, number of parameters mismatch in mutually inductive datatypes"
  pure numParams

private def checkUnsafe (rs : Array ElabHeaderResult) : TermElabM Unit := do
  let isUnsafe := rs[0].view.modifiers.isUnsafe
  for r in rs do
    unless r.view.modifiers.isUnsafe == isUnsafe do
      throwErrorAt r.view.ref "invalid inductive type, cannot mix unsafe and safe declarations in a mutually inductive datatypes"

private def checkLevelNames (views : Array InductiveView) : TermElabM Unit := do
  if views.size > 1 then
    let levelNames := views[0].levelNames
    for view in views do
      unless view.levelNames == levelNames do
        throwErrorAt view.ref "invalid inductive type, universe parameters mismatch in mutually inductive datatypes"
```

```
private def mkTypeFor (r : ElabHeaderResult) : TermElabM Expr := do
  withLCtx r.lctx r.localInsts do
    mkForallFVars r.params r.type

private def throwUnexpectedInductiveType {α} : TermElabM α :=
  throwError "unexpected inductive resulting type"

private def eqvFirstTypeResult (firstType type : Expr) : MetaM Bool :=
  forallTelescopeReducing firstType fun _ firstTypeResult => isDefEq firstTypeResult type

-- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
private partial def checkParamsAndResultType (type firstType : Expr) (numParams : Nat) : TermElabM Unit := do
  try
    forallTelescopeCompatible type firstType numParams fun _ type firstType =>
    forallTelescopeReducing type fun _ type =>
    forallTelescopeReducing firstType fun _ firstType => do
    match type with
    | Expr.sort .. =>
      unless (← isDefEq firstType type) do
        throwError "resulting universe mismatch, given{indentExpr type}\nexpected type{indentExpr firstType}"
    | _ =>
      throwError "unexpected inductive resulting type"
  catch
    | Exception.error ref msg => throw (Exception.error ref m!"invalid mutually inductive types, {msg}")
    | ex => throw ex

-- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
private def checkHeader (r : ElabHeaderResult) (numParams : Nat) (firstType? : Option Expr) : TermElabM Expr := do
  let type ← mkTypeFor r
  match firstType? with
  | none           => pure type
  | some firstType =>
    withRef r.view.ref $ checkParamsAndResultType type firstType numParams
    pure firstType

-- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
private partial def checkHeaders (rs : Array ElabHeaderResult) (numParams : Nat) (i : Nat) (firstType? : Option Expr) : TermElabM Unit :
  if i < rs.size then
    let type ← checkHeader rs[i] numParams firstType?
    checkHeaders rs numParams (i+1) type

private def elabHeader (views : Array InductiveView) : TermElabM (Array ElabHeaderResult) := do
  let rs ← elabHeaderAux views 0 #[]
  if rs.size > 1 then
    checkUnsafe rs
    let numParams ← checkNumParams rs
```

```
    checkHeaders rs numParams 0 none
  return rs

/- Create a local declaration for each inductive type in `rs`, and execute `x params indFVars`, where `params` are the inductive type pa
   `indFVars` are the new local declarations.
   We use the local context/instances and parameters of rs[0].
   Note that this method is executed after we executed `checkHeaders` and established all
   parameters are compatible. -/
private partial def withInductiveLocalDecls {α} (rs : Array ElabHeaderResult) (x : Array Expr → Array Expr → TermElabM α) : TermElabM α
  let namesAndTypes ← rs.mapM fun r => do
    let type ← mkTypeFor r
    pure (r.view.shortDeclName, type)
  let r0     := rs[0]
  let params := r0.params
  withLCtx r0.lctx r0.localInsts $ withRef r0.view.ref do
    let rec loop (i : Nat) (indFVars : Array Expr) := do
      if h : i < namesAndTypes.size then
        let (id, type) := namesAndTypes.get ⟨i, h⟩
        withLocalDeclD id type fun indFVar => loop (i+1) (indFVars.push indFVar)
      else
        x params indFVars
    loop 0 #[]

private def isInductiveFamily (numParams : Nat) (indFVar : Expr) : TermElabM Bool := do
  let indFVarType ← inferType indFVar
  forallTelescopeReducing indFVarType fun xs _ =>
    return xs.size > numParams

/-
  Elaborate constructor types.

  Remark: we check whether the resulting type is correct, and the parameter occurrences are consistent, but
  we currently do not check for:
  - Positivity (it is a rare failure, and the kernel already checks for it).
  - Universe constraints (the kernel checks for it).
-/
private def elabCtors (indFVars : Array Expr) (indFVar : Expr) (params : Array Expr) (r : ElabHeaderResult) : TermElabM (List Constructo
  let indFamily ← isInductiveFamily params.size indFVar
  r.view.ctors.toList.mapM fun ctorView =>
    Term.withAutoBoundImplicit <| Term.elabBinders ctorView.binders.getArgs fun ctorParams =>
      withRef ctorView.ref do
        let rec elabCtorType (k : Expr → TermElabM Constructor) : TermElabM Constructor := do
          match ctorView.type? with
          | none          =>
            if indFamily then
              throwError "constructor resulting type must be specified in inductive family declaration"
            k <| mkAppN indFVar params
```

```
          | some ctorType =>
            let type ← Term.elabType ctorType
            Term.synthesizeSyntheticMVars (mayPostpone := true)
            let type ← instantiateMVars type
            let type ← checkParamOccs type
            forallTelescopeReducing type fun _ resultingType => do
              unless resultingType.getAppFn == indFVar do
                throwError "unexpected constructor resulting type{indentExpr resultingType}"
              unless (← isType resultingType) do
                throwError "unexpected constructor resulting type, type expected{indentExpr resultingType}"
            k type
        elabCtorType fun type => do
          Term.synthesizeSyntheticMVarsNoPostponing
          let ctorParams ← Term.addAutoBoundImplicits ctorParams
          let type ← mkForallFVars ctorParams type
          let type ← mkForallFVars params type
          return { name := ctorView.declName, type := type }
where
  checkParamOccs (ctorType : Expr) : MetaM Expr :=
    let visit (e : Expr) : MetaM TransformStep := do
      let f := e.getAppFn
      if indFVars.contains f then
        let mut args := e.getAppArgs
        unless args.size ≥ params.size do
          throwError "unexpected inductive type occurrence{indentExpr e}"
        for i in [:params.size] do
          let param := params[i]
          let arg := args[i]
          unless (← isDefEq param arg) do
            throwError "inductive datatype parameter mismatch{indentExpr arg}\nexpected{indentExpr param}"
          args := args.set! i param
        return TransformStep.done (mkAppN f args)
      else
        return TransformStep.visit e
    transform ctorType (pre := visit)

/- Convert universe metavariables occurring in the `indTypes` into new parameters.
   Remark: if the resulting inductive datatype has universe metavariables, we will fix it later using
   `inferResultingUniverse`. -/
private def levelMVarToParamAux (indTypes : List InductiveType) : StateRefT Nat TermElabM (List InductiveType) :=
  indTypes.mapM fun indType => do
    let type  ← Term.levelMVarToParam' indType.type
    let ctors ← indType.ctors.mapM fun ctor => do
      let ctorType ← Term.levelMVarToParam' ctor.type
      pure { ctor with type := ctorType }
    pure { indType with ctors := ctors, type := type }
```

```
private def levelMVarToParam (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
  (levelMVarToParamAux indTypes).run' 1

private def getResultingUniverse : List InductiveType → TermElabM Level
  | []           => throwError "unexpected empty inductive declaration"
  | indType :: _ => forallTelescopeReducing indType.type fun _ r => do
    match r with
    | Expr.sort u _ => pure u
    | _             => throwError "unexpected inductive type resulting type"

def tmpIndParam := mkLevelParam `_tmp_ind_univ_param

/--
  Return true if `u` is of the form `?m + k`.
  Return false if `u` does not contain universe metavariables.
  Throw exception otherwise. -/
def shouldInferResultUniverse (u : Level) : TermElabM Bool := do
  let u ← instantiateLevelMVars u
  if u.hasMVar then
    match u.getLevelOffset with
    | Level.mvar mvarId _ => do
      Term.assignLevelMVar mvarId tmpIndParam
      pure true
    | _ =>
      throwError "cannot infer resulting universe level of inductive datatype, given level contains metavariables {mkSort u}, provide un
  else
    pure false

/-
  Auxiliary function for `updateResultingUniverse`
  `accLevelAtCtor u r rOffset us` add `u` components to `us` if they are not already there and it is different from the resulting univer
  If `u` is a `max`, then its components are recursively processed.
  If `u` is a `succ` and `rOffset > 0`, we process the `u`s child using `rOffset-1`.

  This method is used to infer the resulting universe level of an inductive datatype. -/
def accLevelAtCtor : Level → Level → Nat → Array Level → TermElabM (Array Level)
  | Level.max u v _,  r, rOffset,   us => do let us ← accLevelAtCtor u r rOffset us; accLevelAtCtor v r rOffset us
  | Level.imax u v _, r, rOffset,   us => do let us ← accLevelAtCtor u r rOffset us; accLevelAtCtor v r rOffset us
  | Level.zero _,     _, _,         us => pure us
  | Level.succ u _,   r, rOffset+1, us => accLevelAtCtor u r rOffset us
  | u,                r, rOffset,   us =>
    if rOffset == 0 && u == r then pure us
    else if r.occurs u  then throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
    else if rOffset > 0 then throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
    else if us.contains u then pure us
    else pure (us.push u)
```

```
/- Auxiliary function for `updateResultingUniverse` -/
private partial def collectUniversesFromCtorTypeAux (r : Level) (rOffset : Nat) : Nat → Expr → Array Level → TermElabM (Array Level)
  | 0,   Expr.forallE n d b c, us => do
    let u ← getLevel d
    let u ← instantiateLevelMVars u
    let us ← accLevelAtCtor u r rOffset us
    withLocalDecl n c.binderInfo d fun x =>
      let e := b.instantiate1 x
      collectUniversesFromCtorTypeAux r rOffset 0 e us
  | i+1, Expr.forallE n d b c, us => do
    withLocalDecl n c.binderInfo d fun x =>
      let e := b.instantiate1 x
      collectUniversesFromCtorTypeAux r rOffset i e us
  | _, _, us => pure us

/- Auxiliary function for `updateResultingUniverse` -/
private partial def collectUniversesFromCtorType
    (r : Level) (rOffset : Nat) (ctorType : Expr) (numParams : Nat) (us : Array Level) : TermElabM (Array Level) :=
  collectUniversesFromCtorTypeAux r rOffset numParams ctorType us

/- Auxiliary function for `updateResultingUniverse` -/
private partial def collectUniverses (r : Level) (rOffset : Nat) (numParams : Nat) (indTypes : List InductiveType) : TermElabM (Array Le
  let mut us := #[]
  for indType in indTypes do
    for ctor in indType.ctors do
      us ← collectUniversesFromCtorType r rOffset ctor.type numParams us
  return us

def mkResultUniverse (us : Array Level) (rOffset : Nat) : Level :=
  if us.isEmpty && rOffset == 0 then
    levelOne
  else
    let r := Level.mkNaryMax us.toList
    if rOffset == 0 && !r.isZero && !r.isNeverZero then
      (mkLevelMax r levelOne).normalize
    else
      r.normalize

private def updateResultingUniverse (numParams : Nat) (indTypes : List InductiveType) : TermElabM (List InductiveType) := do
  let r ← getResultingUniverse indTypes
  let rOffset : Nat   := r.getOffset
  let r       : Level := r.getLevelOffset
  unless r.isParam do
    throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
  let us ← collectUniverses r rOffset numParams indTypes
  trace[Elab.inductive] "updateResultingUniverse us: {us}, r: {r}, rOffset: {rOffset}"
  let rNew := mkResultUniverse us rOffset
```

```
  let updateLevel (e : Expr) : Expr := e.replaceLevel fun u => if u == tmpIndParam then some rNew else none
  return indTypes.map fun indType =>
    let type := updateLevel indType.type;
    let ctors := indType.ctors.map fun ctor => { ctor with type := updateLevel ctor.type };
    { indType with type := type, ctors := ctors }

register_builtin_option bootstrap.inductiveCheckResultingUniverse : Bool := {
    defValue := true,
    group    := "bootstrap",
    descr    := "by default the `inductive/structure commands report an error if the resulting universe is not zero, but may be zero for
}

def checkResultingUniverse (u : Level) : TermElabM Unit := do
  if bootstrap.inductiveCheckResultingUniverse.get (← getOptions) then
    let u ← instantiateLevelMVars u
    if !u.isZero && !u.isNeverZero then
      throwError "invalid universe polymorphic type, the resultant universe is not Prop (i.e., 0), but it may be Prop for some parameter

private def checkResultingUniverses (indTypes : List InductiveType) : TermElabM Unit := do
  checkResultingUniverse (← getResultingUniverse indTypes)

private def collectUsed (indTypes : List InductiveType) : StateRefT CollectFVars.State MetaM Unit := do
  indTypes.forM fun indType => do
    Term.collectUsedFVars indType.type
    indType.ctors.forM fun ctor =>
      Term.collectUsedFVars ctor.type

private def removeUnused (vars : Array Expr) (indTypes : List InductiveType) : TermElabM (LocalContext × LocalInstances × Array Expr) :=
  let (_, used) ← (collectUsed indTypes).run {}
  Term.removeUnused vars used

private def withUsed {α} (vars : Array Expr) (indTypes : List InductiveType) (k : Array Expr → TermElabM α) : TermElabM α := do
  let (lctx, localInsts, vars) ← removeUnused vars indTypes
  withLCtx lctx localInsts $ k vars

private def updateParams (vars : Array Expr) (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
  indTypes.mapM fun indType => do
    let type ← mkForallFVars vars indType.type
    let ctors ← indType.ctors.mapM fun ctor => do
      let ctorType ← mkForallFVars vars ctor.type
      pure { ctor with type := ctorType }
    pure { indType with type := type, ctors := ctors }

private def collectLevelParamsInInductive (indTypes : List InductiveType) : Array Name := do
  let mut usedParams : CollectLevelParams.State := {}
  for indType in indTypes do
    usedParams := collectLevelParams usedParams indType.type
```

```
    for ctor in indType.ctors do
      usedParams := collectLevelParams usedParams ctor.type
  return usedParams.params

private def mkIndFVar2Const (views : Array InductiveView) (indFVars : Array Expr) (levelNames : List Name) : ExprMap Expr := do
  let levelParams := levelNames.map mkLevelParam;
  let mut m : ExprMap Expr := {}
  for i in [:views.size] do
    let view    := views[i]
    let indFVar := indFVars[i]
    m := m.insert indFVar (mkConst view.declName levelParams)
  return m

/- Remark: `numVars <= numParams`. `numVars` is the number of context `variables` used in the inductive declaration,
   and `numParams` is `numVars` + number of explicit parameters provided in the declaration. -/
private def replaceIndFVarsWithConsts (views : Array InductiveView) (indFVars : Array Expr) (levelNames : List Name)
    (numVars : Nat) (numParams : Nat) (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
  let indFVar2Const := mkIndFVar2Const views indFVars levelNames
  indTypes.mapM fun indType => do
    let ctors ← indType.ctors.mapM fun ctor => do
      let type ← forallBoundedTelescope ctor.type numParams fun params type => do
        let type := type.replace fun e =>
          if !e.isFVar then
            none
          else match indFVar2Const.find? e with
            | none   => none
            | some c => mkAppN c (params.extract 0 numVars)
        mkForallFVars params type
      pure { ctor with type := type }
    pure { indType with ctors := ctors }

abbrev Ctor2InferMod := Std.HashMap Name Bool

private def mkCtor2InferMod (views : Array InductiveView) : Ctor2InferMod := do
  let mut m := {}
  for view in views do
    for ctorView in view.ctors do
      m := m.insert ctorView.declName ctorView.inferMod
  return m

private def applyInferMod (views : Array InductiveView) (numParams : Nat) (indTypes : List InductiveType) : List InductiveType :=
  let ctor2InferMod := mkCtor2InferMod views
  indTypes.map fun indType =>
    let ctors := indType.ctors.map fun ctor =>
      let inferMod := ctor2InferMod.find! ctor.name -- true if `{}` was used
      let ctorType := ctor.type.inferImplicit numParams !inferMod
      { ctor with type := ctorType }
```

```
    { indType with ctors := ctors }

private def mkAuxConstructions (views : Array InductiveView) : TermElabM Unit := do
  let env ← getEnv
  let hasEq   := env.contains ``Eq
  let hasHEq  := env.contains ``HEq
  let hasUnit := env.contains ``PUnit
  let hasProd := env.contains ``Prod
  for view in views do
    let n := view.declName
    mkRecOn n
    if hasUnit then mkCasesOn n
    if hasUnit && hasEq && hasHEq then mkNoConfusion n
    if hasUnit && hasProd then mkBelow n
    if hasUnit && hasProd then mkIBelow n
  for view in views do
    let n := view.declName;
    if hasUnit && hasProd then mkBRecOn n
    if hasUnit && hasProd then mkBInductionOn n

private def mkInductiveDecl (vars : Array Expr) (views : Array InductiveView) : TermElabM Unit := do
  let view0 := views[0]
  let scopeLevelNames ← Term.getLevelNames
  checkLevelNames views
  let allUserLevelNames := view0.levelNames
  let isUnsafe          := view0.modifiers.isUnsafe
  withRef view0.ref <| Term.withLevelNames allUserLevelNames do
    let rs ← elabHeader views
    withInductiveLocalDecls rs fun params indFVars => do
      let numExplicitParams := params.size
      let mut indTypes := #[]
      for i in [:views.size] do
        let indFVar := indFVars[i]
        let r       := rs[i]
        let type  ← mkForallFVars params r.type
        let ctors ← elabCtors indFVars indFVar params r
        indTypes := indTypes.push { name := r.view.declName, type := type, ctors := ctors : InductiveType }
      let indTypes := indTypes.toList
      Term.synthesizeSyntheticMVarsNoPostponing
      let u ← getResultingUniverse indTypes
      let inferLevel ← shouldInferResultUniverse u
      withUsed vars indTypes fun vars => do
        let numVars   := vars.size
        let numParams := numVars + numExplicitParams
        let indTypes ← updateParams vars indTypes
        let indTypes ← levelMVarToParam indTypes
        let indTypes ← if inferLevel then updateResultingUniverse numParams indTypes else checkResultingUniverses indTypes; pure indType
```

```
        let usedLevelNames := collectLevelParamsInInductive indTypes
        match sortDeclLevelParams scopeLevelNames allUserLevelNames usedLevelNames with
        | Except.error msg      => throwError msg
        | Except.ok levelParams => do
          let indTypes ← replaceIndFVarsWithConsts views indFVars levelParams numVars numParams indTypes
          let indTypes := applyInferMod views numParams indTypes
          let decl := Declaration.inductDecl levelParams numParams indTypes isUnsafe
          Term.ensureNoUnassignedMVars decl
          addDecl decl
          mkAuxConstructions views
          -- We need to invoke `applyAttributes` because `class` is implemented as an attribute.
          for view in views do
            Term.applyAttributesAt view.declName view.modifiers.attrs AttributeApplicationTime.afterTypeChecking

private def applyDerivingHandlers (views : Array InductiveView) : CommandElabM Unit := do
  let mut processed : NameSet := {}
  for view in views do
    for classView in view.derivingClasses do
      let className := classView.className
      unless processed.contains className do
        processed := processed.insert className
        let mut declNames := #[]
        for view in views do
          if view.derivingClasses.any fun classView => classView.className == className then
            declNames := declNames.push view.declName
        classView.applyHandlers declNames

def elabInductiveViews (views : Array InductiveView) : CommandElabM Unit := do
  let view0 := views[0]
  let ref := view0.ref
  runTermElabM view0.declName fun vars => withRef ref do
    mkInductiveDecl vars views
    mkSizeOfInstances view0.declName
  applyDerivingHandlers views

end Lean.Elab.Command
::::::::::::::
Elab/InfoTree.lean
::::::::::::::
/-
Copyright (c) 2020 Wojciech Nawrocki. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.

Authors: Wojciech Nawrocki, Leonardo de Moura
-/
import Lean.Data.Position
import Lean.Expr
```

```
import Lean.Message
import Lean.Data.Json
import Lean.Meta.Basic
import Lean.Meta.PPGoal

namespace Lean.Elab

open Std (PersistentArray PersistentArray.empty PersistentHashMap)

/- Context after executing `liftTermElabM`.
   Note that the term information collected during elaboration may contain metavariables, and their
   assignments are stored at `mctx`. -/
structure ContextInfo where
  env           : Environment
  fileMap       : FileMap
  mctx          : MetavarContext := {}
  options       : Options        := {}
  currNamespace : Name           := Name.anonymous
  openDecls     : List OpenDecl  := []
  deriving Inhabited

structure TermInfo where
  lctx : LocalContext -- The local context when the term was elaborated.
  expr : Expr
  stx  : Syntax
  deriving Inhabited

structure CommandInfo where
  stx : Syntax
  deriving Inhabited

inductive CompletionInfo where
  | dot (termInfo : TermInfo) (field? : Option Syntax) (expectedType? : Option Expr)
  | id (stx : Syntax) (id : Name) (danglingDot : Bool) (lctx : LocalContext) (expectedType? : Option Expr)
  | namespaceId (stx : Syntax)
  | option (stx : Syntax)
  | endSection (stx : Syntax) (scopeNames : List String)
  | tactic (stx : Syntax) (goals : List MVarId)
  -- TODO `import`

def CompletionInfo.stx : CompletionInfo → Syntax
  | dot i .. => i.stx
  | id stx .. => stx
  | namespaceId stx => stx
  | option stx => stx
  | endSection stx .. => stx
  | tactic stx .. => stx
```

```

structure FieldInfo where
  name : Name
  lctx : LocalContext
  val  : Expr
  stx  : Syntax
  deriving Inhabited

/- We store the list of goals before and after the execution of a tactic.
   We also store the metavariable context at each time since, we want to unassigned metavariables
   at tactic execution time to be displayed as `?m...`. -/
structure TacticInfo where
  mctxBefore  : MetavarContext
  goalsBefore : List MVarId
  stx         : Syntax
  mctxAfter   : MetavarContext
  goalsAfter  : List MVarId
  deriving Inhabited

structure MacroExpansionInfo where
  lctx   : LocalContext -- The local context when the macro was expanded.
  before : Syntax
  after  : Syntax
  deriving Inhabited

inductive Info where
  | ofTacticInfo (i : TacticInfo)
  | ofTermInfo (i : TermInfo)
  | ofCommandInfo (i : CommandInfo)
  | ofMacroExpansionInfo (i : MacroExpansionInfo)
  | ofFieldInfo (i : FieldInfo)
  | ofCompletionInfo (i : CompletionInfo)
  deriving Inhabited

inductive InfoTree where
  | context (i : ContextInfo) (t : InfoTree) -- The context object is created by `liftTermElabM` at `Command.lean`
  | node (i : Info) (children : PersistentArray InfoTree) -- The children contains information for nested term elaboration and tactic ev
  | ofJson (j : Json) -- For user data
  | hole (mvarId : MVarId) -- The elaborator creates holes (aka metavariables) for tactics and postponed terms
  deriving Inhabited

partial def InfoTree.findInfo? (p : Info → Bool) (t : InfoTree) : Option InfoTree :=
  match t with
  | context _ t => findInfo? p t
  | node i ts   =>
    if p i then
      some t
```

```
    else
      ts.findSome? (findInfo? p)
  | _ => none

structure InfoState where
  enabled    : Bool := false
  assignment : PersistentHashMap MVarId InfoTree := {} -- map from holeId to InfoTree
  trees      : PersistentArray InfoTree := {}
  deriving Inhabited

class MonadInfoTree (m : Type → Type)  where
  getInfoState    : m InfoState
  modifyInfoState : (InfoState → InfoState) → m Unit

export MonadInfoTree (getInfoState modifyInfoState)

instance [MonadLift m n] [MonadInfoTree m] : MonadInfoTree n where
  getInfoState      := liftM (getInfoState : m _)
  modifyInfoState f := liftM (modifyInfoState f : m _)

partial def InfoTree.substitute (tree : InfoTree) (assignment : PersistentHashMap MVarId InfoTree) : InfoTree :=
  match tree with
  | node i c => node i <| c.map (substitute · assignment)
  | context i t => context i (substitute t assignment)
  | ofJson j => ofJson j
  | hole id  => match assignment.find? id with
    | none      => hole id
    | some tree => substitute tree assignment

def ContextInfo.runMetaM (info : ContextInfo) (lctx : LocalContext) (x : MetaM α) : IO α := do
  let x := x.run { lctx := lctx } { mctx := info.mctx }
  let ((a, _), _) ← x.toIO { options := info.options, currNamespace := info.currNamespace, openDecls := info.openDecls } { env := info.e
  return a

def ContextInfo.toPPContext (info : ContextInfo) (lctx : LocalContext) : PPContext :=
  { env  := info.env, mctx := info.mctx, lctx := lctx,
    opts := info.options, currNamespace := info.currNamespace, openDecls := info.openDecls }

def ContextInfo.ppSyntax (info : ContextInfo) (lctx : LocalContext) (stx : Syntax) : IO Format := do
  ppTerm (info.toPPContext lctx) stx

private def formatStxRange (ctx : ContextInfo) (stx : Syntax) : Format := do
  let pos    := stx.getPos?.getD 0
  let endPos := stx.getTailPos?.getD pos
  return f!"{fmtPos pos stx.getHeadInfo}-{fmtPos endPos stx.getTailInfo}"
where fmtPos pos info :=
    let pos := format <| ctx.fileMap.toPosition pos
```

```
    match info with
    | SourceInfo.original ..  => pos
    | _                       => f!"{pos}†"

def TermInfo.runMetaM (info : TermInfo) (ctx : ContextInfo) (x : MetaM α) : IO α :=
  ctx.runMetaM info.lctx x

def TermInfo.format (ctx : ContextInfo) (info : TermInfo) : IO Format := do
  info.runMetaM ctx do
    return f!"{← Meta.ppExpr info.expr} : {← Meta.ppExpr (← Meta.inferType info.expr)} @ {formatStxRange ctx info.stx}"

def CompletionInfo.format (ctx : ContextInfo) (info : CompletionInfo) : IO Format :=
  match info with
  | CompletionInfo.dot i (expectedType? := expectedType?) .. => return f!"[.] {← i.format ctx} : {expectedType?}"
  | CompletionInfo.id stx _ _ lctx expectedType? => ctx.runMetaM lctx do return f!"[.] {stx} : {expectedType?} @ {formatStxRange ctx inf
  | _ => return f!"[.] {info.stx} @ {formatStxRange ctx info.stx}"

def CommandInfo.format (ctx : ContextInfo) (info : CommandInfo) : IO Format := do
  return f!"command @ {formatStxRange ctx info.stx}"

def FieldInfo.format (ctx : ContextInfo) (info : FieldInfo) : IO Format := do
  ctx.runMetaM info.lctx do
    return f!"{info.name} : {← Meta.ppExpr (← Meta.inferType info.val)} := {← Meta.ppExpr info.val} @ {formatStxRange ctx info.stx}"

def ContextInfo.ppGoals (ctx : ContextInfo) (goals : List MVarId) : IO Format :=
  if goals.isEmpty then
    return "no goals"
  else
    ctx.runMetaM {} (return Std.Format.prefixJoin "\n" (← goals.mapM Meta.ppGoal))

def TacticInfo.format (ctx : ContextInfo) (info : TacticInfo) : IO Format := do
  let ctxB := { ctx with mctx := info.mctxBefore }
  let ctxA := { ctx with mctx := info.mctxAfter }
  let goalsBefore ← ctxB.ppGoals info.goalsBefore
  let goalsAfter  ← ctxA.ppGoals info.goalsAfter
  return f!"Tactic @ {formatStxRange ctx info.stx}\nbefore {goalsBefore}\nafter {goalsAfter}"

def MacroExpansionInfo.format (ctx : ContextInfo) (info : MacroExpansionInfo) : IO Format := do
  let before ← ctx.ppSyntax info.lctx info.before
  let after  ← ctx.ppSyntax info.lctx info.after
  return f!"Macro expansion\n{before}\n===>\n{after}"

def Info.format (ctx : ContextInfo) : Info → IO Format
  | ofTacticInfo i         => i.format ctx
  | ofTermInfo i           => i.format ctx
  | ofCommandInfo i        => i.format ctx
  | ofMacroExpansionInfo i => i.format ctx
```

```
  | ofFieldInfo i          => i.format ctx
  | ofCompletionInfo i     => i.format ctx

/--
  Helper function for propagating the tactic metavariable context to its children nodes.
  We need this function because we preserve `TacticInfo` nodes during backtracking *and* their
  children. Moreover, we backtrack the metavariable context to undo metavariable assignments.
  `TacticInfo` nodes save the metavariable context before/after the tactic application, and
  can be pretty printed without any extra information. This is not the case for `TermInfo` nodes.
  Without this function, the formatting method would often fail when processing `TermInfo` nodes
  that are children of `TacticInfo` nodes that have been preserved during backtracking.
  Saving the metavariable context at `TermInfo` nodes is also not a good option because
  at `TermInfo` creation time, the metavariable context often miss information, e.g.,
  a TC problem has not been resolved, a postponed subterm has not been elaborated, etc.

  See `Term.SavedState.restore`.
-/
def Info.updateContext? : Option ContextInfo → Info → Option ContextInfo
  | some ctx, ofTacticInfo i => some { ctx with mctx := i.mctxAfter }
  | ctx?, _ => ctx?

partial def InfoTree.format (tree : InfoTree) (ctx? : Option ContextInfo := none) : IO Format := do
  match tree with
  | ofJson j    => return toString j
  | hole id     => return toString id
  | context i t => format t i
  | node i cs   => match ctx? with
    | none => return "<context-not-available>"
    | some ctx =>
      let fmt ← i.format ctx
      if cs.size == 0 then
        return fmt
      else
        let ctx? := i.updateContext? ctx?
        return f!"{fmt}{Std.Format.nestD <| Std.Format.prefixJoin "\n" (← cs.toList.mapM fun c => format c ctx?)}"

section
variable [Monad m] [MonadInfoTree m]

@[inline] private def modifyInfoTrees (f : PersistentArray InfoTree → PersistentArray InfoTree) : m Unit :=
  modifyInfoState fun s => { s with trees := f s.trees }

private def getResetInfoTrees : m (PersistentArray InfoTree) := do
  let trees := (← getInfoState).trees
  modifyInfoTrees fun _ => {}
  return trees
```

```
def pushInfoTree (t : InfoTree) : m Unit := do
  if (← getInfoState).enabled then
    modifyInfoTrees fun ts => ts.push t

def pushInfoLeaf (t : Info) : m Unit := do
  if (← getInfoState).enabled then
    pushInfoTree <| InfoTree.node (children := {}) t

def addCompletionInfo (info : CompletionInfo) : m Unit := do
  pushInfoLeaf <| Info.ofCompletionInfo info

def resolveGlobalConstNoOverloadWithInfo [MonadResolveName m] [MonadEnv m] [MonadError m] (stx : Syntax) (id := stx.getId) : m Name := d
  let n ← resolveGlobalConstNoOverload id
  if (← getInfoState).enabled then
    pushInfoLeaf <| Info.ofTermInfo { lctx := LocalContext.empty, expr := (← mkConstWithLevelParams n), stx := stx }
  return n

def resolveGlobalConstWithInfos [MonadResolveName m] [MonadEnv m] [MonadError m] (stx : Syntax) (id := stx.getId) : m (List Name) := do
  let ns ← resolveGlobalConst id
  if (← getInfoState).enabled then
    for n in ns do
      pushInfoLeaf <| Info.ofTermInfo { lctx := LocalContext.empty, expr := (← mkConstWithLevelParams n), stx := stx }
  return ns

def mkInfoNode (info : Info) : m Unit := do
  if (← getInfoState).enabled then
    modifyInfoTrees fun ts => PersistentArray.empty.push <| InfoTree.node info ts

@[inline] def withInfoContext' [MonadFinally m] (x : m α) (mkInfo : α → m (Sum Info MVarId)) : m α := do
  if (← getInfoState).enabled then
    let treesSaved ← getResetInfoTrees
    Prod.fst <$> MonadFinally.tryFinally' x fun a? => do
      match a? with
      | none   => modifyInfoTrees fun _ => treesSaved
      | some a =>
        let info ← mkInfo a
        modifyInfoTrees fun trees =>
          match info with
          | Sum.inl info  => treesSaved.push <| InfoTree.node info trees
          | Sum.inr mvaId => treesSaved.push <| InfoTree.hole mvaId
  else
    x

@[inline] def withInfoTreeContext [MonadFinally m] (x : m α) (mkInfoTree : PersistentArray InfoTree → m InfoTree) : m α := do
  if (← getInfoState).enabled then
    let treesSaved ← getResetInfoTrees
    Prod.fst <$> MonadFinally.tryFinally' x fun _ => do
```

```
      let st    ← getInfoState
      let tree  ← mkInfoTree st.trees
      modifyInfoTrees fun _ => treesSaved.push tree
  else
    x

@[inline] def withInfoContext [MonadFinally m] (x : m α) (mkInfo : m Info) : m α := do
  withInfoTreeContext x (fun trees => do return InfoTree.node (← mkInfo) trees)

def getInfoHoleIdAssignment? (mvarId : MVarId) : m (Option InfoTree) :=
  return (← getInfoState).assignment[mvarId]

def assignInfoHoleId (mvarId : MVarId) (infoTree : InfoTree) : m Unit := do
  assert! (← getInfoHoleIdAssignment? mvarId).isNone
  modifyInfoState fun s => { s with assignment := s.assignment.insert mvarId infoTree }
end

def withMacroExpansionInfo [MonadFinally m] [Monad m] [MonadInfoTree m] [MonadLCtx m] (before after : Syntax) (x : m α) : m α :=
  let mkInfo : m Info := do
    return Info.ofMacroExpansionInfo {
      lctx   := (← getLCtx)
      before := before
      after  := after
    }
  withInfoContext x mkInfo

@[inline] def withInfoHole [MonadFinally m] [Monad m] [MonadInfoTree m] (mvarId : MVarId) (x : m α) : m α := do
  if (← getInfoState).enabled then
    let treesSaved ← getResetInfoTrees
    Prod.fst <$> MonadFinally.tryFinally' x fun a? => modifyInfoState fun s =>
      if s.trees.size > 0 then
        { s with trees := treesSaved, assignment := s.assignment.insert mvarId s.trees[s.trees.size - 1] }
      else
        { s with trees := treesSaved }
  else
    x

def enableInfoTree [MonadInfoTree m] (flag := true) : m Unit :=
  modifyInfoState fun s => { s with enabled := flag }

def getInfoTrees [MonadInfoTree m] [Monad m] : m (PersistentArray InfoTree) :=
  return (← getInfoState).trees

end Lean.Elab
::::::::::::::
Elab/LetRec.lean
::::::::::::::
```

```
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Attributes
import Lean.Elab.Binders
import Lean.Elab.DeclModifiers
import Lean.Elab.SyntheticMVars
import Lean.Elab.DeclarationRange

namespace Lean.Elab.Term
open Meta

structure LetRecDeclView where
  ref           : Syntax
  attrs         : Array Attribute
  shortDeclName : Name
  declName      : Name
  numParams     : Nat
  type          : Expr
  mvar          : Expr -- auxiliary metavariable used to lift the 'let rec'
  valStx        : Syntax

structure LetRecView where
  decls     : Array LetRecDeclView
  body      : Syntax

/-  group ("let " >> nonReservedSymbol "rec ") >> sepBy1 (group (optional «attributes» >> letDecl)) ", " >> "; " >> termParser -/
private def mkLetRecDeclView (letRec : Syntax) : TermElabM LetRecView := do
  let decls ← letRec[1][0].getSepArgs.mapM fun (attrDeclStx : Syntax) => do
    let docStr? ← expandOptDocComment? attrDeclStx[0]
    let attrOptStx := attrDeclStx[1]
    let attrs ← if attrOptStx.isNone then pure #[] else elabDeclAttrs attrOptStx[0]
    let decl := attrDeclStx[2][0]
    if decl.isOfKind `Lean.Parser.Term.letPatDecl then
      throwErrorAt decl "patterns are not allowed in 'let rec' expressions"
    else if decl.isOfKind `Lean.Parser.Term.letIdDecl || decl.isOfKind `Lean.Parser.Term.letEqnsDecl then
      let shortDeclName := decl[0].getId
      let currDeclName? ← getDeclName?
      let declName := currDeclName?.getD Name.anonymous ++ shortDeclName
      checkNotAlreadyDeclared declName
      applyAttributesAt declName attrs AttributeApplicationTime.beforeElaboration
      addDocString' declName docStr?
      addAuxDeclarationRanges declName decl decl[0]
      let binders := decl[1].getArgs
      let typeStx := expandOptType decl decl[2]
```

```
      let (type, numParams) ← elabBinders binders fun xs => do
          let type ← elabType typeStx
          registerCustomErrorIfMVar type typeStx "failed to infer 'let rec' declaration type"
          let type ← mkForallFVars xs type
          pure (type, xs.size)
      let mvar ← mkFreshExprMVar type MetavarKind.syntheticOpaque
      let valStx ←
        if decl.isOfKind `Lean.Parser.Term.letIdDecl then
          pure decl[4]
        else
          liftMacroM $ expandMatchAltsIntoMatch decl decl[3]
      pure {
        ref           := decl,
        attrs         := attrs,
        shortDeclName := shortDeclName,
        declName      := declName,
        numParams     := numParams,
        type          := type,
        mvar          := mvar,
        valStx        := valStx
        : LetRecDeclView }
    else
      throwUnsupportedSyntax
  pure {
    decls := decls,
    body  := letRec[3]
  }

private partial def withAuxLocalDecls {α} (views : Array LetRecDeclView) (k : Array Expr → TermElabM α) : TermElabM α :=
  let rec loop (i : Nat) (fvars : Array Expr) : TermElabM α :=
    if h : i < views.size then
      let view := views.get ⟨i, h⟩
      withLocalDeclD view.shortDeclName view.type fun fvar => loop (i+1) (fvars.push fvar)
    else
      k fvars
  loop 0 #[]

private def elabLetRecDeclValues (view : LetRecView) : TermElabM (Array Expr) :=
  view.decls.mapM fun view => do
    forallBoundedTelescope view.type view.numParams fun xs type =>
       withDeclName view.declName do
         let value ← elabTermEnsuringType view.valStx type
         mkLambdaFVars xs value

private def registerLetRecsToLift (views : Array LetRecDeclView) (fvars : Array Expr) (values : Array Expr) : TermElabM Unit := do
  let letRecsToLiftCurr := (← get).letRecsToLift
  for view in views do
```

```
    if letRecsToLiftCurr.any fun toLift => toLift.declName == view.declName then
      withRef view.ref do
        throwError "'{view.declName}' has already been declared"
  let lctx ← getLCtx
  let localInsts ← getLocalInstances
  let toLift := views.mapIdx fun i view => {
    ref            := view.ref,
    fvarId         := fvars[i].fvarId!,
    attrs          := view.attrs,
    shortDeclName  := view.shortDeclName,
    declName       := view.declName,
    lctx           := lctx,
    localInstances := localInsts,
    type           := view.type,
    val            := values[i],
    mvarId         := view.mvar.mvarId!
    : LetRecToLift }
  modify fun s => { s with letRecsToLift := toLift.toList ++ s.letRecsToLift }

@[builtinTermElab «letrec»] def elabLetRec : TermElab := fun stx expectedType? => do
  let view ← mkLetRecDeclView stx
  withAuxLocalDecls view.decls fun fvars => do
    let values ← elabLetRecDeclValues view
    let body ← elabTermEnsuringType view.body expectedType?
    registerLetRecsToLift view.decls fvars values
    let mvars := view.decls.map (·.mvar)
    pure $ mkAppN (← mkLambdaFVars fvars body) mvars

end Lean.Elab.Term
::::::::::::::
Elab/Level.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.LevelDefEq
import Lean.Elab.Exception
import Lean.Elab.Log
import Lean.Elab.AutoBound

namespace Lean.Elab.Level

structure Context where
  options           : Options
  ref               : Syntax
```

```
  autoBoundImplicit : Bool

structure State where
  ngen       : NameGenerator
  mctx       : MetavarContext
  levelNames : List Name

abbrev LevelElabM := ReaderT Context (EStateM Exception State)

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

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

instance : AddMessageContext LevelElabM where
  addMessageContext msg := pure msg

instance : MonadNameGenerator LevelElabM where
  getNGen := return (← get).ngen

  setNGen ngen := modify fun s => { s with ngen := ngen }

def mkFreshLevelMVar : LevelElabM Level := do
  let mvarId ← mkFreshId
  modify fun s => { s with mctx := s.mctx.addLevelMVarDecl mvarId }
  return mkLevelMVar mvarId

register_builtin_option maxUniverseOffset : Nat := {
  defValue := 32
  descr    := "maximum universe level offset"
}

private def checkUniverseOffset [Monad m] [MonadError m] [MonadOptions m] (n : Nat) : m Unit := do
  let max := maxUniverseOffset.get (← getOptions)
  unless n <= max do
    throwError "maximum universe level offset threshold ({max}) has been reached, you can increase the limit using option `set_option ma

partial def elabLevel (stx : Syntax) : LevelElabM Level := withRef stx do
  let kind := stx.getKind
  if kind == `Lean.Parser.Level.paren then
    elabLevel (stx.getArg 1)
  else if kind == `Lean.Parser.Level.max then
    let args := stx.getArg 1 |>.getArgs
    args[:args.size - 1].foldrM (init := ← elabLevel args.back) fun stx lvl =>
      return mkLevelMax' (← elabLevel stx) lvl
```

```
  else if kind == `Lean.Parser.Level.imax then
    let args := stx.getArg 1 |>.getArgs
    args[:args.size - 1].foldrM (init := ← elabLevel args.back) fun stx lvl =>
      return mkLevelIMax' (← elabLevel stx) lvl
  else if kind == `Lean.Parser.Level.hole then
    mkFreshLevelMVar
  else if kind == numLitKind then
    match stx.isNatLit? with
    | some val => checkUniverseOffset val; return Level.ofNat val
    | none     => throwIllFormedSyntax
  else if kind == identKind then
    let paramName := stx.getId
    unless (← get).levelNames.contains paramName do
      if (← read).autoBoundImplicit && isValidAutoBoundLevelName paramName then
        modify fun s => { s with levelNames := paramName :: s.levelNames }
      else
        throwError "unknown universe level '{paramName}'"
    return mkLevelParam paramName
  else if kind == `Lean.Parser.Level.addLit then
    let lvl ← elabLevel (stx.getArg 0)
    match stx.getArg 2 |>.isNatLit? with
    | some val => checkUniverseOffset val; return lvl.addOffset val
    | none     => throwIllFormedSyntax
  else
    throwError "unexpected universe level syntax kind"

end Lean.Elab.Level
::::::::::::::
Elab/Log.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Util
import Lean.Util.Sorry
import Lean.Elab.Exception

namespace Lean.Elab

class MonadFileMap (m : Type → Type) where
  getFileMap  : m FileMap

export MonadFileMap (getFileMap)

class MonadLog (m : Type → Type) extends MonadFileMap m where
```

```
  getRef       : m Syntax
  getFileName  : m String
  logMessage   : Message → m Unit

export MonadLog (getFileName logMessage)

instance (m n) [MonadLift m n] [MonadLog m] : MonadLog n where
  getRef      := liftM (MonadLog.getRef : m _)
  getFileMap  := liftM (getFileMap : m _)
  getFileName := liftM (getFileName : m _)
  logMessage  := fun msg => liftM (logMessage msg : m _ )

variable {m : Type → Type} [Monad m] [MonadLog m] [AddMessageContext m]

def getRefPos : m String.Pos := do
  let ref ← MonadLog.getRef
  return ref.getPos?.getD 0

def getRefPosition : m Position := do
  let fileMap ← getFileMap
  return fileMap.toPosition (← getRefPos)

def logAt (ref : Syntax) (msgData : MessageData) (severity : MessageSeverity := MessageSeverity.error): m Unit :=
  unless severity == MessageSeverity.error && msgData.hasSyntheticSorry do
    let ref    := replaceRef ref (← MonadLog.getRef)
    let pos    := ref.getPos?.getD 0
    let endPos := ref.getTailPos?.getD pos
    let fileMap ← getFileMap
    let msgData ← addMessageContext msgData
    logMessage { fileName := (← getFileName), pos := fileMap.toPosition pos, endPos := fileMap.toPosition endPos, data := msgData, sever

def logErrorAt (ref : Syntax) (msgData : MessageData) : m Unit :=
  logAt ref msgData MessageSeverity.error

def logWarningAt (ref : Syntax) (msgData : MessageData) : m Unit :=
  logAt ref msgData MessageSeverity.warning

def logInfoAt (ref : Syntax) (msgData : MessageData) : m Unit :=
  logAt ref msgData MessageSeverity.information

def log (msgData : MessageData) (severity : MessageSeverity := MessageSeverity.error): m Unit := do
  let ref ← MonadLog.getRef
  logAt ref msgData severity

def logError (msgData : MessageData) : m Unit :=
  log msgData MessageSeverity.error
```

```
def logWarning (msgData : MessageData) : m Unit :=
  log msgData MessageSeverity.warning

def logInfo (msgData : MessageData) : m Unit :=
  log msgData MessageSeverity.information

def logException [MonadLiftT IO m] (ex : Exception) : m Unit := do
  match ex with
  | Exception.error ref msg => logErrorAt ref msg
  | Exception.internal id _ =>
    unless isAbortExceptionId id do
      let name ← id.getName
      logError m!"internal exception: {name}"

def logTrace (cls : Name) (msgData : MessageData) : m Unit := do
  logInfo (MessageData.tagged cls m!"[{cls}] {msgData}")

@[inline] def trace [MonadOptions m] (cls : Name) (msg : Unit → MessageData) : m Unit := do
  if checkTraceOption (← getOptions) cls then
    logTrace cls (msg ())

def logDbgTrace [MonadOptions m] (msg : MessageData) : m Unit := do
  trace `Elab.debug fun _ => msg

def logUnknownDecl (declName : Name) : m Unit :=
  logError m!"unknown declaration '{declName}'"

end Lean.Elab
::::::::::::::
Elab/Match.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.CollectFVars
import Lean.Meta.Match.MatchPatternAttr
import Lean.Meta.Match.Match
import Lean.Meta.SortLocalDecls
import Lean.Meta.GeneralizeVars
import Lean.Elab.SyntheticMVars
import Lean.Elab.App
import Lean.Parser.Term

namespace Lean.Elab.Term
open Meta
```

```
open Lean.Parser.Term

/- This modules assumes "match"-expressions use the following syntax.

```lean
def matchDiscr := leading_parser optional (try (ident >> checkNoWsBefore "no space before ':'" >> ":")) >> termParser

def «match» := leading_parser:leadPrec "match " >> sepBy1 matchDiscr ", " >> optType >> " with " >> matchAlts
```
-/

structure MatchAltView where
  ref   : Syntax
  patterns : Array Syntax
  rhs   : Syntax
  deriving Inhabited

private def expandSimpleMatch (stx discr lhsVar rhs : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  let newStx ← `(let $lhsVar := $discr; $rhs)
  withMacroExpansion stx newStx <| elabTerm newStx expectedType?

private def elabDiscrsWitMatchType (discrStxs : Array Syntax) (matchType : Expr) (expectedType : Expr) : TermElabM (Array Expr × Bool) :=
  let mut discrs := #[]
  let mut i := 0
  let mut matchType := matchType
  let mut isDep := false
  for discrStx in discrStxs do
    i := i + 1
    matchType ← whnf matchType
    match matchType with
    | Expr.forallE _ d b _ =>
      let discr ← fullApproxDefEq <| elabTermEnsuringType discrStx[1] d
      trace[Elab.match] "discr #{i} {discr} : {d}"
      if b.hasLooseBVars then
        isDep := true
      matchType ← b.instantiate1 discr
      discrs := discrs.push discr
    | _ =>
      throwError "invalid type provided to match-expression, function type with arity #{discrStxs.size} expected"
  pure (discrs, isDep)

private def mkUserNameFor (e : Expr) : TermElabM Name := do
  match e with
  /- Remark: we use `mkFreshUserName` to make sure we don't add a variable to the local context that can be resolved to `e`. -/
  | Expr.fvar fvarId _ => mkFreshUserName ((← getLocalDecl fvarId).userName)
  | _   => mkFreshBinderName

```

```lean
/-- Return true iff `n` is an auxiliary variable created by `expandNonAtomicDiscrs?` -/
def isAuxDiscrName (n : Name) : Bool :=
  n.hasMacroScopes && n.eraseMacroScopes == `_discr

/- We treat `@x` as atomic to avoid unnecessary extra local declarations from being
   inserted into the local context. Recall that `expandMatchAltsIntoMatch` uses `@` modifier.
   Thus this is kind of discriminant is quite common.

   Remark: if the discriminat is `Systax.missing`, we abort the elaboration of the `match`-expression.
   This can happen due to error recovery. Example
   ```
   example : (p ∨ p) → p := fun h => match
   ```
   If we don't abort, the elaborator loops because we will keep trying to expand
   ```
   match
   ```
   into
   ```
   let d := <Syntax.missing>; match
   ```
   Recall that `Syntax.setArg stx i arg` is a no-op when `i` is out-of-bounds. -/
def isAtomicDiscr? (discr : Syntax) : TermElabM (Option Expr) := do
  match discr with
  | `($x:ident)  => isLocalIdent? x
  | `(@$x:ident) => isLocalIdent? x
  | _ => if discr.isMissing then throwAbortTerm else return none

-- See expandNonAtomicDiscrs?
private def elabAtomicDiscr (discr : Syntax) : TermElabM Expr := do
  let term := discr[1]
  match (← isAtomicDiscr? term) with
  | some e@(Expr.fvar fvarId _) =>
    let localDecl ← getLocalDecl fvarId
    if !isAuxDiscrName localDecl.userName then
      pure e -- it is not an auxiliary local created by `expandNonAtomicDiscrs?`
    else
      pure localDecl.value
  | _ => throwErrorAt discr "unexpected discriminant"

structure ElabMatchTypeAndDiscsResult where
  discrs    : Array Expr
  matchType : Expr
  /- `true` when performing dependent elimination. We use this to decide whether we optimize the "match unit" case.
     See `isMatchUnit?`. -/
  isDep     : Bool
  alts      : Array MatchAltView
```

```
private def elabMatchTypeAndDiscrs (discrStxs : Array Syntax) (matchOptType : Syntax) (matchAltViews : Array MatchAltView) (expectedType
    : TermElabM ElabMatchTypeAndDiscsResult := do
  let numDiscrs := discrStxs.size
  if matchOptType.isNone then
    let rec loop (i : Nat) (discrs : Array Expr) (matchType : Expr) (isDep : Bool) (matchAltViews : Array MatchAltView) := do
      match i with
      | 0   => return { discrs := discrs.reverse, matchType := matchType, isDep := isDep, alts := matchAltViews }
      | i+1 =>
        let discrStx := discrStxs[i]
        let discr ← elabAtomicDiscr discrStx
        let discr ← instantiateMVars discr
        let discrType ← inferType discr
        let discrType ← instantiateMVars discrType
        let matchTypeBody ← kabstract matchType discr
        let isDep := isDep || matchTypeBody.hasLooseBVars
        let userName ← mkUserNameFor discr
        if discrStx[0].isNone then
          loop i (discrs.push discr) (Lean.mkForall userName BinderInfo.default discrType matchTypeBody) isDep matchAltViews
        else
          let identStx := discrStx[0][0]
          withLocalDeclD userName discrType fun x => do
            let eqType ← mkEq discr x
            withLocalDeclD identStx.getId eqType fun h => do
              let matchTypeBody := matchTypeBody.instantiate1 x
              let matchType ← mkForallFVars #[x, h] matchTypeBody
              let refl ← mkEqRefl discr
              let discrs := (discrs.push refl).push discr
              let matchAltViews := matchAltViews.map fun altView =>
                { altView with patterns := altView.patterns.insertAt (i+1) identStx }
              loop i discrs matchType isDep matchAltViews
    loop discrStxs.size (discrs := #[]) (isDep := false) expectedType matchAltViews
  else
    let matchTypeStx := matchOptType[0][1]
    let matchType ← elabType matchTypeStx
    let (discrs, isDep) ← elabDiscrsWitMatchType discrStxs matchType expectedType
    return { discrs := discrs, matchType := matchType, isDep := isDep, alts := matchAltViews }

def expandMacrosInPatterns (matchAlts : Array MatchAltView) : MacroM (Array MatchAltView) := do
  matchAlts.mapM fun matchAlt => do
    let patterns ← matchAlt.patterns.mapM expandMacros
    pure { matchAlt with patterns := patterns }

private def getMatchGeneralizing? : Syntax → Option Bool
  | `(match (generalizing := true)  $discrs,* $[: $ty?]? with $alts:matchAlt*) => some true
  | `(match (generalizing := false) $discrs,* $[: $ty?]? with $alts:matchAlt*) => some false
  | _ => none
```

```
/- Given `stx` a match-expression, return its alternatives. -/
private def getMatchAlts : Syntax → Array MatchAltView
  | `(match $[$gen]? $discrs,* $[: $ty?]? with $alts:matchAlt*) =>
    alts.filterMap fun alt => match alt with
      | `(matchAltExpr| | $patterns,* => $rhs) => some {
          ref      := alt,
          patterns := patterns,
          rhs      := rhs
        }
      | _ => none
  | _ => #[]

/--
  Auxiliary annotation used to mark terms marked with the "inaccessible" annotation `.(t)` and
  `_` in patterns. -/
def mkInaccessible (e : Expr) : Expr :=
  mkAnnotation `_inaccessible e

def inaccessible? (e : Expr) : Option Expr :=
  annotation? `_inaccessible e

inductive PatternVar where
  | localVar     (userName : Name)
  -- anonymous variables (`_`) are encoded using metavariables
  | anonymousVar (mvarId   : MVarId)

instance : ToString PatternVar := ⟨fun
  | PatternVar.localVar x          => toString x
  | PatternVar.anonymousVar mvarId => s!"?m{mvarId}"⟩

builtin_initialize Parser.registerBuiltinNodeKind `MVarWithIdKind

/--
  Create an auxiliary Syntax node wrapping a fresh metavariable id.
  We use this kind of Syntax for representing `_` occurring in patterns.
  The metavariables are created before we elaborate the patterns into `Expr`s. -/
private def mkMVarSyntax : TermElabM Syntax := do
  let mvarId ← mkFreshId
  return Syntax.node `MVarWithIdKind #[Syntax.node mvarId #[]]

/-- Given a syntax node constructed using `mkMVarSyntax`, return its MVarId -/
private def getMVarSyntaxMVarId (stx : Syntax) : MVarId :=
  stx[0].getKind

/--
  The elaboration function for `Syntax` created using `mkMVarSyntax`.
```

```
  It just converts the metavariable id wrapped by the Syntax into an `Expr`. -/
@[builtinTermElab MVarWithIdKind] def elabMVarWithIdKind : TermElab := fun stx expectedType? =>
  return mkInaccessible <| mkMVar (getMVarSyntaxMVarId stx)

@[builtinTermElab inaccessible] def elabInaccessible : TermElab := fun stx expectedType? => do
  let e ← elabTerm stx[1] expectedType?
  return mkInaccessible e

/-
  Patterns define new local variables.
  This module collect them and preprocess `_` occurring in patterns.
  Recall that an `_` may represent anonymous variables or inaccessible terms
  that are implied by typing constraints. Thus, we represent them with fresh named holes `?x`.
  After we elaborate the pattern, if the metavariable remains unassigned, we transform it into
  a regular pattern variable. Otherwise, it becomes an inaccessible term.

  Macros occurring in patterns are expanded before the `collectPatternVars` method is executed.
  The following kinds of Syntax are handled by this module
  - Constructor applications
  - Applications of functions tagged with the `[matchPattern]` attribute
  - Identifiers
  - Anonymous constructors
  - Structure instances
  - Inaccessible terms
  - Named patterns
  - Tuple literals
  - Type ascriptions
  - Literals: num, string and char
-/
namespace CollectPatternVars

structure State where
  found     : NameSet := {}
  vars      : Array PatternVar := #[]

abbrev M := StateRefT State TermElabM

private def throwCtorExpected {α} : M α :=
  throwError "invalid pattern, constructor or constant marked with '[matchPattern]' expected"

private def getNumExplicitCtorParams (ctorVal : ConstructorVal) : TermElabM Nat :=
  forallBoundedTelescope ctorVal.type ctorVal.numParams fun ps _ => do
    let mut result := 0
    for p in ps do
      let localDecl ← getLocalDecl p.fvarId!
      if localDecl.binderInfo.isExplicit then
        result := result+1
```

```
  pure result

private def throwInvalidPattern {α} : M α :=
  throwError "invalid pattern"

/-
An application in a pattern can be

1- A constructor application
   The elaborator assumes fields are accessible and inductive parameters are not accessible.

2- A regular application `(f ...)` where `f` is tagged with `[matchPattern]`.
   The elaborator assumes implicit arguments are not accessible and explicit ones are accessible.
-/

structure Context where
  funId         : Syntax
  ctorVal?      : Option ConstructorVal -- It is `some`, if constructor application
  explicit      : Bool
  ellipsis      : Bool
  paramDecls    : Array (Name × BinderInfo) -- parameters names and binder information
  paramDeclIdx  : Nat := 0
  namedArgs     : Array NamedArg
  args          : List Arg
  newArgs       : Array Syntax := #[]
  deriving Inhabited

private def isDone (ctx : Context) : Bool :=
  ctx.paramDeclIdx ≥ ctx.paramDecls.size

private def finalize (ctx : Context) : M Syntax := do
  if ctx.namedArgs.isEmpty && ctx.args.isEmpty then
    let fStx ← `(@$(ctx.funId):ident)
    return Syntax.mkApp fStx ctx.newArgs
  else
    throwError "too many arguments"

private def isNextArgAccessible (ctx : Context) : Bool :=
  let i := ctx.paramDeclIdx
  match ctx.ctorVal? with
  | some ctorVal => i ≥ ctorVal.numParams -- For constructor applications only fields are accessible
  | none =>
    if h : i < ctx.paramDecls.size then
      -- For `[matchPattern]` applications, only explicit parameters are accessible.
      let d := ctx.paramDecls.get ⟨i, h⟩
      d.2.isExplicit
    else
```

```
    false

private def getNextParam (ctx : Context) : (Name × BinderInfo) × Context :=
  let i := ctx.paramDeclIdx
  let d := ctx.paramDecls[i]
  (d, { ctx with paramDeclIdx := ctx.paramDeclIdx + 1 })

private def processVar (idStx : Syntax) : M Syntax := do
  unless idStx.isIdent do
    throwErrorAt idStx "identifier expected"
  let id := idStx.getId
  unless id.eraseMacroScopes.isAtomic do
    throwError "invalid pattern variable, must be atomic"
  if (← get).found.contains id then
    throwError "invalid pattern, variable '{id}' occurred more than once"
  modify fun s => { s with vars := s.vars.push (PatternVar.localVar id), found := s.found.insert id }
  return idStx

private def nameToPattern : Name → TermElabM Syntax
  | Name.anonymous => `(Name.anonymous)
  | Name.str p s _ => do let p ← nameToPattern p; `(Name.str $p $(quote s) _)
  | Name.num p n _ => do let p ← nameToPattern p; `(Name.num $p $(quote n) _)

private def quotedNameToPattern (stx : Syntax) : TermElabM Syntax :=
  match stx[0].isNameLit? with
  | some val => nameToPattern val
  | none     => throwIllFormedSyntax

private def doubleQuotedNameToPattern (stx : Syntax) : TermElabM Syntax := do
  match stx[1].isNameLit? with
  | some val => nameToPattern (← resolveGlobalConstNoOverloadWithInfo stx[1] val)
  | none     => throwIllFormedSyntax

partial def collect (stx : Syntax) : M Syntax := withRef stx <| withFreshMacroScope do
  let k := stx.getKind
  if k == identKind then
    processId stx
  else if k == ``Lean.Parser.Term.app then
    processCtorApp stx
  else if k == ``Lean.Parser.Term.anonymousCtor then
    let elems ← stx[1].getArgs.mapSepElemsM collect
    return stx.setArg 1 <| mkNullNode elems
  else if k == ``Lean.Parser.Term.structInst then
    /-
    ```
    leading_parser "{" >> optional (atomic (termParser >> " with "))
                >> manyIndent (group (structInstField >> optional ", "))
```
```

```
              >> optional ".."
              >> optional (" : " >> termParser)
              >> " }"
  ```
  -/
  let withMod := stx[1]
  unless withMod.isNone do
    throwErrorAt withMod "invalid struct instance pattern, 'with' is not allowed in patterns"
  let fields ← stx[2].getArgs.mapM fun p => do
      -- p is of the form (group (structInstField >> optional ", "))
      let field := p[0]
      -- leading_parser structInstLVal >> " := " >> termParser
      let newVal ← collect field[2]
      let field := field.setArg 2 newVal
      pure <| field.setArg 0 field
  return stx.setArg 2 <| mkNullNode fields
else if k == ``Lean.Parser.Term.hole then
  let r ← mkMVarSyntax
  modify fun s => { s with vars := s.vars.push <| PatternVar.anonymousVar <| getMVarSyntaxMVarId r }
  return r
else if k == ``Lean.Parser.Term.paren then
  let arg := stx[1]
  if arg.isNone then
    return stx -- `()`
  else
    let t := arg[0]
    let s := arg[1]
    if s.isNone || s[0].getKind == ``Lean.Parser.Term.typeAscription then
      -- Ignore `s`, since it empty or it is a type ascription
      let t ← collect t
      let arg := arg.setArg 0 t
      return stx.setArg 1 arg
    else
      -- Tuple literal is a constructor
      let t ← collect t
      let arg := arg.setArg 0 t
      let tupleTail := s[0]
      let tupleTailElems := tupleTail[1].getArgs
      let tupleTailElems ← tupleTailElems.mapSepElemsM collect
      let tupleTail := tupleTail.setArg 1 <| mkNullNode tupleTailElems
      let s   := s.setArg 0 tupleTail
      let arg   := arg.setArg 1 s
      return stx.setArg 1 arg
else if k == ``Lean.Parser.Term.explicitUniv then
  processCtor stx[0]
else if k == ``Lean.Parser.Term.namedPattern then
  /- Recall that
```
```

```
      def namedPattern := check... >> trailing_parser "@" >> termParser -/
    let id := stx[0]
    discard <| processVar id
    let pat := stx[2]
    let pat ← collect pat
    `(_root_.namedPattern $id $pat)
  else if k == ``Lean.Parser.Term.inaccessible then
    return stx
  else if k == strLitKind then
    return stx
  else if k == numLitKind then
    return stx
  else if k == scientificLitKind then
    return stx
  else if k == charLitKind then
    return stx
  else if k == ``Lean.Parser.Term.quotedName then
    /- Quoted names have an elaboration function associated with them, and they will not be macro expanded.
      Note that macro expansion is not a good option since it produces a term using the smart constructors `Name.mkStr`, `Name.mkNum`
      instead of the constructors `Name.str` and `Name.num` -/
    quotedNameToPattern stx
  else if k == ``Lean.Parser.Term.doubleQuotedName then
    /- Similar to previous case -/
    doubleQuotedNameToPattern stx
  else if k == choiceKind then
    throwError "invalid pattern, notation is ambiguous"
  else
    throwInvalidPattern

where

  processCtorApp (stx : Syntax) : M Syntax := do
    let (f, namedArgs, args, ellipsis) ← expandApp stx true
    processCtorAppCore f namedArgs args ellipsis

  processCtor (stx : Syntax) : M Syntax := do
    processCtorAppCore stx #[] #[] false

  /- Check whether `stx` is a pattern variable or constructor-like (i.e., constructor or constant tagged with `[matchPattern]` attribute
  processId (stx : Syntax) : M Syntax := do
    match (← resolveId? stx "pattern") with
    | none   => processVar stx
    | some f => match f with
      | Expr.const fName _ _ =>
        match (← getEnv).find? fName with
        | some (ConstantInfo.ctorInfo _) => processCtor stx
        | some _ =>
```

```
        if hasMatchPatternAttribute (← getEnv) fName then
          processCtor stx
        else
          processVar stx
      | none => throwCtorExpected
    | _ => processVar stx

pushNewArg (accessible : Bool) (ctx : Context) (arg : Arg) : M Context := do
  match arg with
  | Arg.stx stx =>
    let stx ← if accessible then collect stx else pure stx
    return { ctx with newArgs := ctx.newArgs.push stx }
  | _ => unreachable!

processExplicitArg (accessible : Bool) (ctx : Context) : M Context := do
  match ctx.args with
  | [] =>
    if ctx.ellipsis then
      pushNewArg accessible ctx (Arg.stx (← `(_)))
    else
      throwError "explicit parameter is missing, unused named arguments {ctx.namedArgs.map fun narg => narg.name}"
  | arg::args =>
    pushNewArg accessible { ctx with args := args } arg

processImplicitArg (accessible : Bool) (ctx : Context) : M Context := do
  if ctx.explicit then
    processExplicitArg accessible ctx
  else
    pushNewArg accessible ctx (Arg.stx (← `(_)))

processCtorAppContext (ctx : Context) : M Syntax := do
  if isDone ctx then
    finalize ctx
  else
    let accessible := isNextArgAccessible ctx
    let (d, ctx)   := getNextParam ctx
    match ctx.namedArgs.findIdx? fun namedArg => namedArg.name == d.1 with
    | some idx =>
      let arg := ctx.namedArgs[idx]
      let ctx := { ctx with namedArgs := ctx.namedArgs.eraseIdx idx }
      let ctx ← pushNewArg accessible ctx arg.val
      processCtorAppContext ctx
    | none =>
      let ctx ← match d.2 with
        | BinderInfo.implicit     => processImplicitArg accessible ctx
        | BinderInfo.instImplicit => processImplicitArg accessible ctx
        | _                       => processExplicitArg accessible ctx
```

```
      processCtorAppContext ctx

  processCtorAppCore (f : Syntax) (namedArgs : Array NamedArg) (args : Array Arg) (ellipsis : Bool) : M Syntax := do
    let args := args.toList
    let (fId, explicit) ← match f with
      | `($fId:ident)  => pure (fId, false)
      | `(@$fId:ident) => pure (fId, true)
      | _              => throwError "identifier expected"
    let some (Expr.const fName _ _) ← resolveId? fId "pattern" | throwCtorExpected
    let fInfo ← getConstInfo fName
    let paramDecls ← forallTelescopeReducing fInfo.type fun xs _ => xs.mapM fun x => do
      let d ← getFVarLocalDecl x
      return (d.userName, d.binderInfo)
    match fInfo with
    | ConstantInfo.ctorInfo val =>
      processCtorAppContext
        { funId := fId, explicit := explicit, ctorVal? := val, paramDecls := paramDecls, namedArgs := namedArgs, args := args, ellipsis
    | _ =>
      if hasMatchPatternAttribute (← getEnv) fName then
        processCtorAppContext
          { funId := fId, explicit := explicit, ctorVal? := none, paramDecls := paramDecls, namedArgs := namedArgs, args := args, ellips
      else
        throwCtorExpected

def main (alt : MatchAltView) : M MatchAltView := do
  let patterns ← alt.patterns.mapM fun p => do
    trace[Elab.match] "collecting variables at pattern: {p}"
    collect p
  return { alt with patterns := patterns }

end CollectPatternVars

private def collectPatternVars (alt : MatchAltView) : TermElabM (Array PatternVar × MatchAltView) := do
  let (alt, s) ← (CollectPatternVars.main alt).run {}
  return (s.vars, alt)

/- Return the pattern variables in the given pattern.
  Remark: this method is not used here, but in other macros (e.g., at `Do.lean`). -/
def getPatternVars (patternStx : Syntax) : TermElabM (Array PatternVar) := do
  let patternStx ← liftMacroM <| expandMacros patternStx
  let (_, s) ← (CollectPatternVars.collect patternStx).run {}
  return s.vars

def getPatternsVars (patterns : Array Syntax) : TermElabM (Array PatternVar) := do
  let collect : CollectPatternVars.M Unit := do
    for pattern in patterns do
      discard <| CollectPatternVars.collect (← liftMacroM <| expandMacros pattern)
```

```
  let (_, s) ← collect.run {}
  return s.vars

/- We convert the collected `PatternVar`s intro `PatternVarDecl` -/
inductive PatternVarDecl where
  /- For `anonymousVar`, we create both a metavariable and a free variable. The free variable is used as an assignment for the metavaria
     when it is not assigned during pattern elaboration. -/
  | anonymousVar (mvarId : MVarId) (fvarId : FVarId)
  | localVar     (fvarId : FVarId)

private partial def withPatternVars {α} (pVars : Array PatternVar) (k : Array PatternVarDecl → TermElabM α) : TermElabM α :=
  let rec loop (i : Nat) (decls : Array PatternVarDecl) := do
    if h : i < pVars.size then
      match pVars.get ⟨i, h⟩ with
      | PatternVar.anonymousVar mvarId =>
        let type ← mkFreshTypeMVar
        let userName ← mkFreshBinderName
        withLocalDecl userName BinderInfo.default type fun x =>
          loop (i+1) (decls.push (PatternVarDecl.anonymousVar mvarId x.fvarId!))
      | PatternVar.localVar userName   =>
        let type ← mkFreshTypeMVar
        withLocalDecl userName BinderInfo.default type fun x =>
          loop (i+1) (decls.push (PatternVarDecl.localVar x.fvarId!))
    else
      /- We must create the metavariables for `PatternVar.anonymousVar` AFTER we create the new local decls using `withLocalDecl`.
         Reason: their scope must include the new local decls since some of them are assigned by typing constraints. -/
      decls.forM fun decl => match decl with
        | PatternVarDecl.anonymousVar mvarId fvarId => do
          let type ← inferType (mkFVar fvarId)
          discard <| mkFreshExprMVarWithId mvarId type
        | _ => pure ()
      k decls
  loop 0 #[]

/-
Remark: when performing dependent pattern matching, we often had to write code such as

```lean
def Vec.map' (f : α → β) (xs : Vec α n) : Vec β n :=
  match n, xs with
  | _, nil       => nil
  | _, cons a as => cons (f a) (map' f as)
```
We had to include `n` and the `_`s because the type of `xs` depends on `n`.
Moreover, `nil` and `cons a as` have different types.
This was quite tedious. So, we have implemented an automatic "discriminant refinement procedure".
The procedure is based on the observation that we get a type error whenenver we forget to include `_`s
```

```
and the indices a discriminant depends on. So, we catch the exception, check whether the type of the discriminant
is an indexed family, and add their indices as new discriminants.

The current implementation, adds indices as they are found, and does not
try to "sort" the new discriminants.

If the refinement process fails, we report the original error message.
-/

/- Auxiliary structure for storing an type mismatch exception when processing the
   pattern #`idx` of some alternative. -/
structure PatternElabException where
  ex  : Exception
  idx : Nat

private def elabPatterns (patternStxs : Array Syntax) (matchType : Expr) : ExceptT PatternElabException TermElabM (Array Expr × Expr) :=
  withReader (fun ctx => { ctx with implicitLambda := false }) do
    let mut patterns  := #[]
    let mut matchType := matchType
    for idx in [:patternStxs.size] do
      let patternStx := patternStxs[idx]
      matchType ← whnf matchType
      match matchType with
      | Expr.forallE _ d b _ =>
          let pattern ←
            try
              liftM <| withSynthesize <| withoutErrToSorry <| elabTermEnsuringType patternStx d
            catch ex =>
              -- Wrap the type mismatch exception for the "discriminant refinement" feature.
              throwThe PatternElabException { ex := ex, idx := idx }
          matchType := b.instantiate1 pattern
          patterns  := patterns.push pattern
      | _ => throwError "unexpected match type"
    return (patterns, matchType)

def finalizePatternDecls (patternVarDecls : Array PatternVarDecl) : TermElabM (Array LocalDecl) := do
  let mut decls := #[]
  for pdecl in patternVarDecls do
    match pdecl with
    | PatternVarDecl.localVar fvarId =>
      let decl ← getLocalDecl fvarId
      let decl ← instantiateLocalDeclMVars decl
      decls := decls.push decl
    | PatternVarDecl.anonymousVar mvarId fvarId =>
       let e ← instantiateMVars (mkMVar mvarId);
       trace[Elab.match] "finalizePatternDecls: mvarId: {mvarId} := {e}, fvar: {mkFVar fvarId}"
       match e with
```

```
      | Expr.mvar newMVarId _ =>
        /- Metavariable was not assigned, or assigned to another metavariable. So,
           we assign to the auxiliary free variable we created at `withPatternVars` to `newMVarId`. -/
        assignExprMVar newMVarId (mkFVar fvarId)
        trace[Elab.match] "finalizePatternDecls: {mkMVar newMVarId} := {mkFVar fvarId}"
        let decl ← getLocalDecl fvarId
        let decl ← instantiateLocalDeclMVars decl
        decls := decls.push decl
      | _ => pure ()
  /- We perform a topological sort (dependecies) on `decls` because the pattern elaboration process may produce a sequence where a decla
  sortLocalDecls decls

open Meta.Match (Pattern Pattern.var Pattern.inaccessible Pattern.ctor Pattern.as Pattern.val Pattern.arrayLit AltLHS MatcherResult)

namespace ToDepElimPattern

structure State where
  found      : NameSet := {}
  localDecls : Array LocalDecl
  newLocals  : NameSet := {}

abbrev M := StateRefT State TermElabM

private def alreadyVisited (fvarId : FVarId) : M Bool := do
  let s ← get
  return s.found.contains fvarId

private def markAsVisited (fvarId : FVarId) : M Unit :=
  modify fun s => { s with found := s.found.insert fvarId }

private def throwInvalidPattern {α} (e : Expr) : M α :=
  throwError "invalid pattern {indentExpr e}"

/- Create a new LocalDecl `x` for the metavariable `mvar`, and return `Pattern.var x` -/
private def mkLocalDeclFor (mvar : Expr) : M Pattern := do
  let mvarId := mvar.mvarId!
  let s ← get
  match (← getExprMVarAssignment? mvarId) with
  | some val => return Pattern.inaccessible val
  | none =>
    let fvarId ← mkFreshId
    let type   ← inferType mvar
    /- HACK: `fvarId` is not in the scope of `mvarId`
       If this generates problems in the future, we should update the metavariable declarations. -/
    assignExprMVar mvarId (mkFVar fvarId)
    let userName ← mkFreshBinderName
    let newDecl := LocalDecl.cdecl arbitrary fvarId userName type BinderInfo.default;
```

```
    modify fun s =>
      { s with
        newLocals  := s.newLocals.insert fvarId,
        localDecls :=
        match s.localDecls.findIdx? fun decl => mvar.occurs decl.type with
        | none   => s.localDecls.push newDecl -- None of the existing declarations depend on `mvar`
        | some i => s.localDecls.insertAt i newDecl }
    return Pattern.var fvarId

partial def main (e : Expr) : M Pattern := do
  let isLocalDecl (fvarId : FVarId) : M Bool := do
    return (← get).localDecls.any fun d => d.fvarId == fvarId
  let mkPatternVar (fvarId : FVarId) (e : Expr) : M Pattern := do
    if (← alreadyVisited fvarId) then
      return Pattern.inaccessible e
    else
      markAsVisited fvarId
      return Pattern.var e.fvarId!
  let mkInaccessible (e : Expr) : M Pattern := do
    match e with
    | Expr.fvar fvarId _ =>
      if (← isLocalDecl fvarId) then
        mkPatternVar fvarId e
      else
        return Pattern.inaccessible e
    | _ =>
      return Pattern.inaccessible e
  match inaccessible? e with
  | some t => mkInaccessible t
  | none =>
    match e.arrayLit? with
    | some (α, lits) =>
      return Pattern.arrayLit α (← lits.mapM main)
    | none =>
      if e.isAppOfArity `namedPattern 3 then
        let p ← main <| e.getArg! 2
        match e.getArg! 1 with
        | Expr.fvar fvarId _ => return Pattern.as fvarId p
        | _                  => throwError "unexpected occurrence of auxiliary declaration 'namedPattern'"
      else if e.isNatLit || e.isStringLit || e.isCharLit then
        return Pattern.val e
      else if e.isFVar then
        let fvarId := e.fvarId!
        unless (← isLocalDecl fvarId) do
          throwInvalidPattern e
        mkPatternVar fvarId e
      else if e.isMVar then
```

```
      mkLocalDeclFor e
    else
      let newE ← whnf e
      if newE != e then
        main newE
      else matchConstCtor e.getAppFn (fun _ => throwInvalidPattern e) fun v us => do
        let args := e.getAppArgs
        unless args.size == v.numParams + v.numFields do
          throwInvalidPattern e
        let params := args.extract 0 v.numParams
        let fields := args.extract v.numParams args.size
        let fields ← fields.mapM main
        return Pattern.ctor v.name us params.toList fields.toList

end ToDepElimPattern

def withDepElimPatterns {α} (localDecls : Array LocalDecl) (ps : Array Expr) (k : Array LocalDecl → Array Pattern → TermElabM α) : TermE
  let (patterns, s) ← (ps.mapM ToDepElimPattern.main).run { localDecls := localDecls }
  let localDecls ← s.localDecls.mapM fun d => instantiateLocalDeclMVars d
  /- toDepElimPatterns may have added new localDecls. Thus, we must update the local context before we execute `k` -/
  let lctx ← getLCtx
  let lctx := localDecls.foldl (fun (lctx : LocalContext) d => lctx.erase d.fvarId) lctx
  let lctx := localDecls.foldl (fun (lctx : LocalContext) d => lctx.addDecl d) lctx
  withTheReader Meta.Context (fun ctx => { ctx with lctx := lctx }) do
    k localDecls patterns

private def withElaboratedLHS {α} (ref : Syntax) (patternVarDecls : Array PatternVarDecl) (patternStxs : Array Syntax) (matchType : Expr
    (k : AltLHS → Expr → TermElabM α) : ExceptT PatternElabException TermElabM α := do
  let (patterns, matchType) ← withSynthesize <| elabPatterns patternStxs matchType
  id (α := TermElabM α) do
    let localDecls ← finalizePatternDecls patternVarDecls
    let patterns ← patterns.mapM (instantiateMVars ·)
    withDepElimPatterns localDecls patterns fun localDecls patterns =>
      k { ref := ref, fvarDecls := localDecls.toList, patterns := patterns.toList } matchType

private def elabMatchAltView (alt : MatchAltView) (matchType : Expr) : ExceptT PatternElabException TermElabM (AltLHS × Expr) := withRef
  let (patternVars, alt) ← collectPatternVars alt
  trace[Elab.match] "patternVars: {patternVars}"
  withPatternVars patternVars fun patternVarDecls => do
    withElaboratedLHS alt.ref patternVarDecls alt.patterns matchType fun altLHS matchType => do
      let rhs ← elabTermEnsuringType alt.rhs matchType
      let xs := altLHS.fvarDecls.toArray.map LocalDecl.toExpr
      let rhs ← if xs.isEmpty then pure <| mkSimpleThunk rhs else mkLambdaFVars xs rhs
      trace[Elab.match] "rhs: {rhs}"
      return (altLHS, rhs)

/--
```

```
  Collect indices for the "discriminant refinement feature". This method is invoked
  when we detect a type mismatch at a pattern #`idx` of some alternative. -/
private def getIndicesToInclude (discrs : Array Expr) (idx : Nat) : TermElabM (Array Expr) := do
  let discrType ← whnfD (← inferType discrs[idx])
  matchConstInduct discrType.getAppFn (fun _ => return #[]) fun info _ => do
    let mut result := #[]
    let args := discrType.getAppArgs
    for arg in args[info.numParams : args.size] do
      unless (← discrs.anyM fun discr => isDefEq discr arg) do
        result := result.push arg
    return result

private partial def elabMatchAltViews (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) : TermElabM (Array Expr ×
  loop discrs matchType altViews none
where
  /-
    "Discriminant refinement" main loop.
    `first?` contains the first error message we found before updated the `discrs`. -/
  loop (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) (first? : Option (SavedState × Exception))
      : TermElabM (Array Expr × Expr × Array (AltLHS × Expr) × Bool) := do
    let s ← saveState
    match ← altViews.mapM (fun alt => elabMatchAltView alt matchType) |>.run with
    | Except.ok alts => return (discrs, matchType, alts, first?.isSome)
    | Except.error { idx := idx, ex := ex } =>
      let indices ← getIndicesToInclude discrs idx
      if indices.isEmpty then
        throwEx (← updateFirst first? ex)
      else
        let first ← updateFirst first? ex
        s.restore
        let indices ← collectDeps indices discrs
        let matchType ←
          try
            updateMatchType indices matchType
          catch ex =>
            throwEx first
        let altViews  ← addWildcardPatterns indices.size altViews
        let discrs    := indices ++ discrs
        loop discrs matchType altViews first

  throwEx {α} (p : SavedState × Exception) : TermElabM α := do
    p.1.restore; throw p.2

  updateFirst (first? : Option (SavedState × Exception)) (ex : Exception) : TermElabM (SavedState × Exception) := do
    match first? with
    | none       => return (← saveState, ex)
    | some first => return first
```

```
containsFVar (es : Array Expr) (fvarId : FVarId) : Bool :=
  es.any fun e => e.isFVar && e.fvarId! == fvarId

/- Update `indices` by including any free variable `x` s.t.
   - Type of some `discr` depends on `x`.
   - Type of `x` depends on some free variable in `indices`.

   If we don't include these extra variables in indices, then
   `updateMatchType` will generate a type incorrect term.
   For example, suppose `discr` contains `h : @HEq α a α b`, and
   `indices` is `#[α, b]`, and `matchType` is `@HEq α a α b → B`.
   `updateMatchType indices matchType` produces the type
   `(α' : Type) → (b : α') → @HEq α' a α' b → B` which is type incorrect
   because we have `a : α`.
   The method `collectDeps` will include `a` into `indices`.

   This method does not handle dependencies among non-free variables.
   We rely on the type checking method `check` at `updateMatchType`. -/
collectDeps (indices : Array Expr) (discrs : Array Expr) : TermElabM (Array Expr) := do
  let mut s : CollectFVars.State := {}
  for discr in discrs do
    s := collectFVars s (← instantiateMVars (← inferType discr))
  let (indicesFVar, indicesNonFVar) := indices.split Expr.isFVar
  let indicesFVar := indicesFVar.map Expr.fvarId!
  let mut toAdd := #[]
  for fvarId in s.fvarSet.toList do
    unless containsFVar discrs fvarId || containsFVar indices fvarId do
      let localDecl ← getLocalDecl fvarId
      let mctx ← getMCtx
      for indexFVarId in indicesFVar do
        if mctx.localDeclDependsOn localDecl indexFVarId then
          toAdd := toAdd.push fvarId
  let lctx ← getLCtx
  let indicesFVar := (indicesFVar ++ toAdd).qsort fun fvarId₁ fvarId₂ =>
    (lctx.get! fvarId₁).index < (lctx.get! fvarId₂).index
  return indicesFVar.map mkFVar ++ indicesNonFVar

updateMatchType (indices : Array Expr) (matchType : Expr) : TermElabM Expr := do
  let matchType ← indices.foldrM (init := matchType) fun index matchType => do
    let indexType ← inferType index
    let matchTypeBody ← kabstract matchType index
    let userName ← mkUserNameFor index
    return Lean.mkForall userName BinderInfo.default indexType matchTypeBody
  check matchType
  return matchType
```

```
  addWildcardPatterns (num : Nat) (altViews : Array MatchAltView) : TermElabM (Array MatchAltView) := do
    let hole := mkHole (← getRef)
    let wildcards := mkArray num hole
    return altViews.map fun altView => { altView with patterns := wildcards ++ altView.patterns }

def mkMatcher (elimName : Name) (matchType : Expr) (numDiscrs : Nat) (lhss : List AltLHS) : TermElabM MatcherResult :=
  Meta.Match.mkMatcher elimName matchType numDiscrs lhss

register_builtin_option match.ignoreUnusedAlts : Bool := {
  defValue := false
  descr := "if true, do not generate error if an alternative is not used"
}

def reportMatcherResultErrors (altLHSS : List AltLHS) (result : MatcherResult) : TermElabM Unit := do
  unless result.counterExamples.isEmpty do
    withHeadRefOnly <| throwError "missing cases:\n{Meta.Match.counterExamplesToMessageData result.counterExamples}"
  unless match.ignoreUnusedAlts.get (← getOptions) || result.unusedAltIdxs.isEmpty do
    let mut i := 0
    for alt in altLHSS do
      if result.unusedAltIdxs.contains i then
        withRef alt.ref do
          logError "redundant alternative"
      i := i + 1

/--
  If `altLHSS + rhss` is encoding `| PUnit.unit => rhs[0]`, return `rhs[0]`
  Otherwise, return none.
-/
private def isMatchUnit? (altLHSS : List Match.AltLHS) (rhss : Array Expr) : MetaM (Option Expr) := do
  assert! altLHSS.length == rhss.size
  match altLHSS with
  | [ { fvarDecls := [], patterns := [ Pattern.ctor `PUnit.unit .. ], .. } ] =>
    /- Recall that for alternatives of the form `| PUnit.unit => rhs`, `rhss[0]` is of the form `fun _ : Unit => b`. -/
    match rhss[0] with
    | Expr.lam _ _ b _ => return if b.hasLooseBVars then none else b
    | _ => return none
  | _ => return none

/--
  "Generalize" variables that depend on the discriminants.

  Remarks and limitations:
  - If `matchType` is a proposition, then we generalize even when the user did not provide `(generalizing := true)`.
    Motivation: users should have control about the actual `match`-expressions in their programs.
  - We currently do not generalize let-decls.
  - We abort generalization if the new `matchType` is type incorrect.
  - Only discriminants that are free variables are considered during specialization.
```

```
  - We "generalize" by adding new discriminants and pattern variables. We do not "clear" the generalized variables,
    but they become inaccessible since they are shadowed by the patterns variables. We assume this is ok since
    this is the exact behavior users would get if they had written it by hand. Recall there is no `clear` in term mode.
-/
private def generalize (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) (generalizing? : Option Bool) : TermElab
  let gen ←
    match generalizing? with
    | some g => pure g
    | _ => isProp matchType
  if !gen then
    return (discrs, matchType, altViews, false)
  else
    let ysFVarIds ← getFVarsToGeneralize discrs
    /- let-decls are currently being ignored by the generalizer. -/
    let ysFVarIds ← ysFVarIds.filterM fun fvarId => return !(← getLocalDecl fvarId).isLet
    if ysFVarIds.isEmpty then
      return (discrs, matchType, altViews, false)
    else
      let ys := ysFVarIds.map mkFVar
      -- trace[Meta.debug] "ys: {ys}, discrs: {discrs}"
      let matchType' ← forallBoundedTelescope matchType discrs.size fun ds type => do
        let type ← mkForallFVars ys type
        let (discrs', ds') := Array.unzip <| Array.zip discrs ds |>.filter fun (di, d) => di.isFVar
        let type := type.replaceFVars discrs' ds'
        mkForallFVars ds type
      -- trace[Meta.debug] "matchType': {matchType'}"
      if (← isTypeCorrect matchType') then
        let discrs := discrs ++ ys
        let altViews ← altViews.mapM fun altView => do
          let patternVars ← getPatternsVars altView.patterns
          -- We traverse backwards because we want to keep the most recent names.
          -- For example, if `ys` contains `#[h, h]`, we want to make sure `mkFreshUsername is applied to the first `h`,
          -- since it is already shadowed by the second.
          let ysUserNames ← ys.foldrM (init := #[]) fun ys ysUserNames => do
            let yDecl ← getLocalDecl ys.fvarId!
            let mut yUserName := yDecl.userName
            if ysUserNames.contains yUserName then
              yUserName ← mkFreshUserName yUserName
            -- Explicitly provided pattern variables shadow `y`
            else if patternVars.any fun | PatternVar.localVar x => x == yUserName | _ => false then
              yUserName ← mkFreshUserName yUserName
            return ysUserNames.push yUserName
          let ysIds ← ysUserNames.reverse.mapM fun n => return mkIdentFrom (← getRef) n
          return { altView with patterns := altView.patterns ++ ysIds }
        return (discrs, matchType', altViews, true)
      else
        return (discrs, matchType, altViews, true)
```

```
private def elabMatchAux (generalizing? : Option Bool) (discrStxs : Array Syntax) (altViews : Array MatchAltView) (matchOptType : Syntax
    : TermElabM Expr := do
  let mut generalizing? := generalizing?
  if !matchOptType.isNone then
    if generalizing? == some true then
      throwError "the '(generalizing := true)' parameter is not supported when the 'match' type is explicitly provided"
    generalizing? := some false
  let (discrs, matchType, altLHSS, isDep, rhss) ← commitIfDidNotPostpone do
    let ⟨discrs, matchType, isDep, altViews⟩ ← elabMatchTypeAndDiscrs discrStxs matchOptType altViews expectedType
    let (discrs, matchType, altViews, gen) ← generalize discrs matchType altViews generalizing?
    let isDep := isDep || gen
    let matchAlts ← liftMacroM <| expandMacrosInPatterns altViews
    trace[Elab.match] "matchType: {matchType}"
    let (discrs, matchType, alts, refined) ← elabMatchAltViews discrs matchType matchAlts
    let isDep := isDep || refined
    /-
     We should not use `synthesizeSyntheticMVarsNoPostponing` here. Otherwise, we will not be
     able to elaborate examples such as:
     ```
     def f (x : Nat) : Option Nat := none

     def g (xs : List (Nat × Nat)) : IO Unit :=
     xs.forM fun x =>
       match f x.fst with
       | _ => pure ()
     ```
     If `synthesizeSyntheticMVarsNoPostponing`, the example above fails at `x.fst` because
     the type of `x` is only available after we proces the last argument of `List.forM`.

     We apply pending default types to make sure we can process examples such as
     ```
     let (a, b) := (0, 0)
     ```
    -/
    synthesizeSyntheticMVarsUsingDefault
    let rhss := alts.map Prod.snd
    let matchType ← instantiateMVars matchType
    let altLHSS ← alts.toList.mapM fun alt => do
      let altLHS ← Match.instantiateAltLHSMVars alt.1
      /- Remark: we try to postpone before throwing an error.
         The combinator `commitIfDidNotPostpone` ensures we backtrack any updates that have been performed.
         The quick-check `waitExpectedTypeAndDiscrs` minimizes the number of scenarios where we have to postpone here.
         Here is an example that passes the `waitExpectedTypeAndDiscrs` test, but postpones here.
         ```
          def bad (ps : Array (Nat × Nat)) : Array (Nat × Nat) :=
            (ps.filter fun (p : Prod _ _) =>
```
```

```
            match p with
            | (x, y) => x == 0)
          ++
          ps
        ```
        When we try to elaborate `fun (p : Prod _ _) => ...` for the first time, we haven't propagated the type of `ps` yet
        because `Array.filter` has type `{α : Type u_1} → (α → Bool) → (as : Array α) → optParam Nat 0 → optParam Nat (Array.size as) →
        However, the partial type annotation `(p : Prod _ _)` makes sure we succeed at the quick-check `waitExpectedTypeAndDiscrs`.
      -/
      withRef altLHS.ref do
        for d in altLHS.fvarDecls do
            if d.hasExprMVar then
            withExistingLocalDecls altLHS.fvarDecls do
              tryPostpone
              throwMVarError m!"invalid match-expression, type of pattern variable '{d.toExpr}' contains metavariables{indentExpr d.type
        for p in altLHS.patterns do
          if p.hasExprMVar then
            withExistingLocalDecls altLHS.fvarDecls do
              tryPostpone
              throwMVarError m!"invalid match-expression, pattern contains metavariables{indentExpr (← p.toExpr)}"
        pure altLHS
    return (discrs, matchType, altLHSS, isDep, rhss)
  if let some r ← if isDep then pure none else isMatchUnit? altLHSS rhss then
    return r
  else
    let numDiscrs := discrs.size
    let matcherName ← mkAuxName `match
    let matcherResult ← mkMatcher matcherName matchType numDiscrs altLHSS
    let motive ← forallBoundedTelescope matchType numDiscrs fun xs matchType => mkLambdaFVars xs matchType
    reportMatcherResultErrors altLHSS matcherResult
    let r := mkApp matcherResult.matcher motive
    let r := mkAppN r discrs
    let r := mkAppN r rhss
    trace[Elab.match] "result: {r}"
    return r

private def getDiscrs (matchStx : Syntax) : Array Syntax :=
  matchStx[2].getSepArgs

private def getMatchOptType (matchStx : Syntax) : Syntax :=
  matchStx[3]

private def expandNonAtomicDiscrs? (matchStx : Syntax) : TermElabM (Option Syntax) :=
  let matchOptType := getMatchOptType matchStx;
  if matchOptType.isNone then do
    let discrs := getDiscrs matchStx;
    let allLocal ← discrs.allM fun discr => Option.isSome <$> isAtomicDiscr? discr[1]
```
```

```
    if allLocal then
      return none
    else
      -- We use `foundFVars` to make sure the discriminants are distinct variables.
      -- See: code for computing "matchType" at `elabMatchTypeAndDiscrs`
      let rec loop (discrs : List Syntax) (discrsNew : Array Syntax) (foundFVars : NameSet) := do
        match discrs with
        | [] =>
          let discrs := Syntax.mkSep discrsNew (mkAtomFrom matchStx ", ");
          pure (matchStx.setArg 2 discrs)
        | discr :: discrs =>
          -- Recall that
          -- matchDiscr := leading_parser optional (ident >> ":") >> termParser
          let term := discr[1]
          let addAux : TermElabM Syntax := withFreshMacroScope do
            let d ← `(_discr);
            unless isAuxDiscrName d.getId do -- Use assertion?
              throwError "unexpected internal auxiliary discriminant name"
            let discrNew := discr.setArg 1 d;
            let r ← loop discrs (discrsNew.push discrNew) foundFVars
            `(let _discr := $term; $r)
          match (← isAtomicDiscr? term) with
          | some x  => if x.isFVar then loop discrs (discrsNew.push discr) (foundFVars.insert x.fvarId!) else addAux
          | none    => addAux
      return some (← loop discrs.toList #[] {})
  else
    -- We do not pull non atomic discriminants when match type is provided explicitly by the user
    return none

private def waitExpectedType (expectedType? : Option Expr) : TermElabM Expr := do
  tryPostponeIfNoneOrMVar expectedType?
  match expectedType? with
    | some expectedType => pure expectedType
    | none              => mkFreshTypeMVar

private def tryPostponeIfDiscrTypeIsMVar (matchStx : Syntax) : TermElabM Unit := do
  -- We don't wait for the discriminants types when match type is provided by user
  if getMatchOptType matchStx |>.isNone then
    let discrs := getDiscrs matchStx
    for discr in discrs do
      let term := discr[1]
      match (← isAtomicDiscr? term) with
      | none   => throwErrorAt discr "unexpected discriminant" -- see `expandNonAtomicDiscrs?
      | some d =>
        let dType ← inferType d
        trace[Elab.match] "discr {d} : {dType}"
        tryPostponeIfMVar dType
```

```lean

/-
We (try to) elaborate a `match` only when the expected type is available.
If the `matchType` has not been provided by the user, we also try to postpone elaboration if the type
of a discriminant is not available. That is, it is of the form `(?m ...)`.
We use `expandNonAtomicDiscrs?` to make sure all discriminants are local variables.
This is a standard trick we use in the elaborator, and it is also used to elaborate structure instances.
Suppose, we are trying to elaborate
```
match g x with
  | ... => ...
```
`expandNonAtomicDiscrs?` converts it intro
```
let _discr := g x
match _discr with
  | ... => ...
```
Thus, at `tryPostponeIfDiscrTypeIsMVar` we only need to check whether the type of `_discr` is not of the form `(?m ...)`.
Note that, the auxiliary variable `_discr` is expanded at `elabAtomicDiscr`.

This elaboration technique is needed to elaborate terms such as:
```lean
xs.filter fun (a, b) => a > b
```
which are syntax sugar for
```lean
List.filter (fun p => match p with | (a, b) => a > b) xs
```
When we visit `match p with | (a, b) => a > b`, we don't know the type of `p` yet.
-/
private def waitExpectedTypeAndDiscrs (matchStx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  tryPostponeIfNoneOrMVar expectedType?
  tryPostponeIfDiscrTypeIsMVar matchStx
  match expectedType? with
  | some expectedType => return expectedType
  | none   => mkFreshTypeMVar

/-
```
leading_parser:leadPrec "match " >> sepBy1 matchDiscr ", " >> optType >> " with " >> matchAlts
```
Remark the `optIdent` must be `none` at `matchDiscr`. They are expanded by `expandMatchDiscr?`.
-/
private def elabMatchCore (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  let expectedType ← waitExpectedTypeAndDiscrs stx expectedType?
  let discrStxs := (getDiscrs stx).map fun d => d
```

```
  let gen?         := getMatchGeneralizing? stx
  let altViews     := getMatchAlts stx
  let matchOptType := getMatchOptType stx
  elabMatchAux gen? discrStxs altViews matchOptType expectedType

private def isPatternVar (stx : Syntax) : TermElabM Bool := do
  match (← resolveId? stx "pattern") with
  | none   => isAtomicIdent stx
  | some f => match f with
    | Expr.const fName _ _ =>
      match (← getEnv).find? fName with
      | some (ConstantInfo.ctorInfo _) => return false
      | some _                         => return !hasMatchPatternAttribute (← getEnv) fName
      | _                              => isAtomicIdent stx
    | _ => isAtomicIdent stx
where
  isAtomicIdent (stx : Syntax) : Bool :=
    stx.isIdent && stx.getId.eraseMacroScopes.isAtomic

-- leading_parser "match " >> sepBy1 termParser ", " >> optType >> " with " >> matchAlts
@[builtinTermElab «match»] def elabMatch : TermElab := fun stx expectedType? => do
  match stx with
  | `(match $discr:term with | $y:ident => $rhs:term) =>
     if (← isPatternVar y) then expandSimpleMatch stx discr y rhs expectedType? else elabMatchDefault stx expectedType?
  | _ => elabMatchDefault stx expectedType?
where
  elabMatchDefault (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
    match (← expandNonAtomicDiscrs? stx) with
    | some stxNew => withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
    | none =>
      let discrs       := getDiscrs stx;
      let matchOptType := getMatchOptType stx;
      if !matchOptType.isNone && discrs.any fun d => !d[0].isNone then
        throwErrorAt matchOptType "match expected type should not be provided when discriminants with equality proofs are used"
      elabMatchCore stx expectedType?

builtin_initialize
  registerTraceClass `Elab.match

-- leading_parser:leadPrec "nomatch " >> termParser
@[builtinTermElab «nomatch»] def elabNoMatch : TermElab := fun stx expectedType? => do
  match stx with
  | `(nomatch $discrExpr) =>
    match ← isLocalIdent? discrExpr with
    | some _ =>
      let expectedType ← waitExpectedType expectedType?
      let discr := Syntax.node ``Lean.Parser.Term.matchDiscr #[mkNullNode, discrExpr]
```

```
      elabMatchAux none #[discr] #[] mkNullNode expectedType
    | _ =>
      let stxNew ← `(let _discr := $discrExpr; nomatch _discr)
      withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
  | _ => throwUnsupportedSyntax

end Lean.Elab.Term
::::::::::::::
Elab/MutualDef.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Term
import Lean.Meta.Closure
import Lean.Meta.Check
import Lean.Elab.Command
import Lean.Elab.DefView
import Lean.Elab.PreDefinition
import Lean.Elab.DeclarationRange

namespace Lean.Elab
open Lean.Parser.Term

/- DefView after elaborating the header. -/
structure DefViewElabHeader where
  ref           : Syntax
  modifiers     : Modifiers
  kind          : DefKind
  shortDeclName : Name
  declName      : Name
  levelNames    : List Name
  numParams     : Nat
  type          : Expr -- including the parameters
  valueStx      : Syntax
  deriving Inhabited

namespace Term

open Meta

private def checkModifiers (m₁ m₂ : Modifiers) : TermElabM Unit := do
  unless m₁.isUnsafe == m₂.isUnsafe do
    throwError "cannot mix unsafe and safe definitions"
  unless m₁.isNoncomputable == m₂.isNoncomputable do
```

```
    throwError "cannot mix computable and non-computable definitions"
  unless m₁.isPartial == m₂.isPartial do
    throwError "cannot mix partial and non-partial definitions"

private def checkKinds (k₁ k₂ : DefKind) : TermElabM Unit := do
  unless k₁.isExample == k₂.isExample do
    throwError "cannot mix examples and definitions" -- Reason: we should discard examples
  unless k₁.isTheorem == k₂.isTheorem do
    throwError "cannot mix theorems and definitions" -- Reason: we will eventually elaborate theorems in `Task`s.

private def check (prevHeaders : Array DefViewElabHeader) (newHeader : DefViewElabHeader) : TermElabM Unit := do
  if newHeader.kind.isTheorem && newHeader.modifiers.isUnsafe then
    throwError "'unsafe' theorems are not allowed"
  if newHeader.kind.isTheorem && newHeader.modifiers.isPartial then
    throwError "'partial' theorems are not allowed, 'partial' is a code generation directive"
  if newHeader.kind.isTheorem && newHeader.modifiers.isNoncomputable then
    throwError "'theorem' subsumes 'noncomputable', code is not generated for theorems"
  if newHeader.modifiers.isNoncomputable && newHeader.modifiers.isUnsafe then
    throwError "'noncomputable unsafe' is not allowed"
  if newHeader.modifiers.isNoncomputable && newHeader.modifiers.isPartial then
    throwError "'noncomputable partial' is not allowed"
  if newHeader.modifiers.isPartial && newHeader.modifiers.isUnsafe then
    throwError "'unsafe' subsumes 'partial'"
  if h : 0 < prevHeaders.size then
    let firstHeader := prevHeaders.get ⟨0, h⟩
    try
      unless newHeader.levelNames == firstHeader.levelNames do
        throwError "universe parameters mismatch"
      checkModifiers newHeader.modifiers firstHeader.modifiers
      checkKinds newHeader.kind firstHeader.kind
    catch
      | Exception.error ref msg => throw (Exception.error ref m!"invalid mutually recursive definitions, {msg}")
      | ex => throw ex
  else
    pure ()

private def registerFailedToInferDefTypeInfo (type : Expr) (ref : Syntax) : TermElabM Unit :=
  registerCustomErrorIfMVar type ref "failed to infer definition type"

private def elabHeaders (views : Array DefView) : TermElabM (Array DefViewElabHeader) := do
  let mut headers := #[]
  for view in views do
    let newHeader ← withRef view.ref do
      let ⟨shortDeclName, declName, levelNames⟩ ← expandDeclId (← getCurrNamespace) (← getLevelNames) view.declId view.modifiers
      addDeclarationRanges declName view.ref
      applyAttributesAt declName view.modifiers.attrs AttributeApplicationTime.beforeElaboration
      withDeclName declName <| withAutoBoundImplicit <| withLevelNames levelNames <|
```

```
      elabBinders view.binders.getArgs fun xs => do
        let refForElabFunType := view.value
        let type ← match view.type? with
          | some typeStx =>
            let type ← elabType typeStx
            registerFailedToInferDefTypeInfo type typeStx
            pure type
          | none =>
            let hole := mkHole refForElabFunType
            let type ← elabType hole
            registerFailedToInferDefTypeInfo type refForElabFunType
            pure type
        Term.synthesizeSyntheticMVarsNoPostponing
        let type ← mkForallFVars xs type
        let type ← mkForallFVars (← read).autoBoundImplicits.toArray type
        let type ← instantiateMVars type
        let xs ← addAutoBoundImplicits xs
        let levelNames ← getLevelNames
        if view.type?.isSome then
          let pendingMVarIds ← getMVars type
          discard <| logUnassignedUsingErrorInfos pendingMVarIds <|
            m!"\nwhen the resulting type of a declaration is explicitly provided, all holes (e.g., `_`) in the header are resolved bef
        let newHeader := {
          ref           := view.ref,
          modifiers     := view.modifiers,
          kind          := view.kind,
          shortDeclName := shortDeclName,
          declName      := declName,
          levelNames    := levelNames,
          numParams     := xs.size,
          type          := type,
          valueStx      := view.value : DefViewElabHeader }
        check headers newHeader
        pure newHeader
    headers := headers.push newHeader
  pure headers

private partial def withFunLocalDecls {α} (headers : Array DefViewElabHeader) (k : Array Expr → TermElabM α) : TermElabM α :=
  let rec loop (i : Nat) (fvars : Array Expr) := do
    if h : i < headers.size then
      let header := headers.get ⟨i, h⟩
      withLocalDecl header.shortDeclName BinderInfo.auxDecl header.type fun fvar => loop (i+1) (fvars.push fvar)
    else
      k fvars
  loop 0 #[]

private def expandWhereDeclsAsStructInst : Macro
```

```
  | `(whereDecls|where $[$decls:letRecDecl$[;]?]*) => do
    let letIdDecls ← decls.mapM fun stx => match stx with
      | `(letRecDecl|$attrs:attributes $decl:letDecl) => Macro.throwErrorAt stx "attributes are 'where' elements are currently not suppo
      | `(letRecDecl|$decl:letPatDecl)  => Macro.throwErrorAt stx "patterns are not allowed here"
      | `(letRecDecl|$decl:letEqnsDecl) => expandLetEqnsDecl decl
      | `(letRecDecl|$decl:letIdDecl)   => pure decl
      | _                               => Macro.throwUnsupported
    let structInstFields ← letIdDecls.mapM fun
      | stx@`(letIdDecl|$id:ident $[$binders]* $[: $ty?]? := $val) => withRef stx do
        let mut val := val
        if let some ty := ty? then
          val ← `(($val : $ty))
        val ← `(fun $[$binders]* => $val:term)
        `(structInstField|$id:ident := $val)
      | _ => Macro.throwUnsupported
    `({ $[$structInstFields,]* })
  | _ => Macro.throwUnsupported

/-
Recall that
```
def declValSimple   := leading_parser " :=\n" >> termParser >> optional Term.whereDecls
def declValEqns   := leading_parser Term.matchAltsWhereDecls
def declVal   := declValSimple <|> declValEqns <|> Term.whereDecls
```
-/
private def declValToTerm (declVal : Syntax) : MacroM Syntax := withRef declVal do
  if declVal.isOfKind `Lean.Parser.Command.declValSimple then
    expandWhereDeclsOpt declVal[2] declVal[1]
  else if declVal.isOfKind `Lean.Parser.Command.declValEqns then
    expandMatchAltsWhereDecls declVal[0]
  else if declVal.isOfKind `Lean.Parser.Term.whereDecls then
    expandWhereDeclsAsStructInst declVal
  else if declVal.isMissing then
    Macro.throwErrorAt declVal "declaration body is missing"
  else
    Macro.throwErrorAt declVal "unexpected declaration body"

private def elabFunValues (headers : Array DefViewElabHeader) : TermElabM (Array Expr) :=
  headers.mapM fun header => withDeclName header.declName $ withLevelNames header.levelNames do
    let valStx ← liftMacroM $ declValToTerm header.valueStx
    forallBoundedTelescope header.type header.numParams fun xs type => do
      let val ← elabTermEnsuringType valStx type
      mkLambdaFVars xs val

private def collectUsed (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
    : StateRefT CollectFVars.State MetaM Unit := do
```

```
  headers.forM fun header => collectUsedFVars header.type
  values.forM collectUsedFVars
  toLift.forM fun letRecToLift => do
    collectUsedFVars letRecToLift.type
    collectUsedFVars letRecToLift.val

private def removeUnusedVars (vars : Array Expr) (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
    : TermElabM (LocalContext × LocalInstances × Array Expr) := do
  let (_, used) ← (collectUsed headers values toLift).run {}
  removeUnused vars used

private def withUsed {α} (vars : Array Expr) (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
    (k : Array Expr → TermElabM α) : TermElabM α := do
  let (lctx, localInsts, vars) ← removeUnusedVars vars headers values toLift
  withLCtx lctx localInsts $ k vars

private def isExample (views : Array DefView) : Bool :=
  views.any (·.kind.isExample)

private def isTheorem (views : Array DefView) : Bool :=
  views.any (·.kind.isTheorem)

private def instantiateMVarsAtHeader (header : DefViewElabHeader) : TermElabM DefViewElabHeader := do
  let type ← instantiateMVars header.type
  pure { header with type := type }

private def instantiateMVarsAtLetRecToLift (toLift : LetRecToLift) : TermElabM LetRecToLift := do
  let type ← instantiateMVars toLift.type
  let val ← instantiateMVars toLift.val
  pure { toLift with type := type, val := val }

private def typeHasRecFun (type : Expr) (funFVars : Array Expr) (letRecsToLift : List LetRecToLift) : Option FVarId :=
  let occ? := type.find? fun e => match e with
    | Expr.fvar fvarId _ => funFVars.contains e || letRecsToLift.any fun toLift => toLift.fvarId == fvarId
    | _ => false
  match occ? with
  | some (Expr.fvar fvarId _) => some fvarId
  | _ => none

private def getFunName (fvarId : FVarId) (letRecsToLift : List LetRecToLift) : TermElabM Name := do
  match (← findLocalDecl? fvarId) with
  | some decl => pure decl.userName
  | none =>
    /- Recall that the FVarId of nested let-recs are not in the current local context. -/
    match letRecsToLift.findSome? fun toLift => if toLift.fvarId == fvarId then some toLift.shortDeclName else none with
    | none   => throwError "unknown function"
    | some n => pure n
```

```lean

/-
Ensures that the of let-rec definition types do not contain functions being defined.
In principle, this test can be improved. We could perform it after we separate the set of functions is strongly connected components.
However, this extra complication doesn't seem worth it.
-/
private def checkLetRecsToLiftTypes (funVars : Array Expr) (letRecsToLift : List LetRecToLift) : TermElabM Unit :=
  letRecsToLift.forM fun toLift =>
    match typeHasRecFun toLift.type funVars letRecsToLift with
    | none        => pure ()
    | some fvarId => do
      let fnName ← getFunName fvarId letRecsToLift
      throwErrorAt toLift.ref "invalid type in 'let rec', it uses '{fnName}' which is being defined simultaneously"

namespace MutualClosure

/- A mapping from FVarId to Set of FVarIds. -/
abbrev UsedFVarsMap := NameMap NameSet

/-
Create the `UsedFVarsMap` mapping that takes the variable id for the mutually recursive functions being defined to the set of
free variables in its definition.

For `mainFVars`, this is just the set of section variables `sectionVars` used.
For nested let-rec functions, we collect their free variables.

Recall that a `let rec` expressions are encoded as follows in the elaborator.
```lean
let rec
  f : A := t,
  g : B := s;
body
```
is encoded as
```lean
let f : A := ?m₁;
let g : B := ?m₂;
body
```
where `?m₁` and `?m₂` are synthetic opaque metavariables. That are assigned by this module.
We may have nested `let rec`s.
```lean
let rec f : A :=
    let rec g : B := t;
    s;
body
```
```

```
is encoded as
```lean
let f : A := ?m₁;
body
```
and the body of `f` is stored the field `val` of a `LetRecToLift`. For the example above,
we would have a `LetRecToLift` containing:
```
{
  mvarId := m₁,
  val    := `(let g : B := ?m₂; body)
  ...
}
```
Note that `g` is not a free variable at `(let g : B := ?m₂; body)`. We recover the fact that
`f` depends on `g` because it contains `m₂`
-/
private def mkInitialUsedFVarsMap (mctx : MetavarContext) (sectionVars : Array Expr) (mainFVarIds : Array FVarId) (letRecsToLift : List
    : UsedFVarsMap := do
  let mut sectionVarSet := {}
  for var in sectionVars do
    sectionVarSet := sectionVarSet.insert var.fvarId!
  let mut usedFVarMap := {}
  for mainFVarId in mainFVarIds do
    usedFVarMap := usedFVarMap.insert mainFVarId sectionVarSet
  for toLift in letRecsToLift do
    let state := Lean.collectFVars {} toLift.val
    let state := Lean.collectFVars state toLift.type
    let mut set := state.fvarSet
    /- toLift.val may contain metavariables that are placeholders for nested let-recs. We should collect the fvarId
       for the associated let-rec because we need this information to compute the fixpoint later. -/
    let mvarIds := (toLift.val.collectMVars {}).result
    for mvarId in mvarIds do
      match letRecsToLift.findSome? fun (toLift : LetRecToLift) => if toLift.mvarId == mctx.getDelayedRoot mvarId then some toLift.fvarI
      | some fvarId => set := set.insert fvarId
      | none   => pure ()
    usedFVarMap := usedFVarMap.insert toLift.fvarId set
  pure usedFVarMap

/-
The let-recs may invoke each other. Example:
```
let rec
  f (x : Nat) := g x + y
  g : Nat → Nat
    | 0   => 1
    | x+1 => f x + z
```
```

````
```
`y` is free variable in `f`, and `z` is a free variable in `g`.
To close `f` and `g`, `y` and `z` must be in the closure of both.
That is, we need to generate the top-level definitions.
```
def f (y z x : Nat) := g y z x + y
def g (y z : Nat) : Nat → Nat
  | 0 => 1
  | x+1 => f y z x + z
```
-/
namespace FixPoint

structure State where
  usedFVarsMap : UsedFVarsMap := {}
  modified     : Bool         := false

abbrev M := ReaderT (List FVarId) $ StateM State

private def isModified : M Bool := do pure (← get).modified
private def resetModified : M Unit := modify fun s => { s with modified := false }
private def markModified : M Unit := modify fun s => { s with modified := true }
private def getUsedFVarsMap : M UsedFVarsMap := do pure (← get).usedFVarsMap
private def modifyUsedFVars (f : UsedFVarsMap → UsedFVarsMap) : M Unit := modify fun s => { s with usedFVarsMap := f s.usedFVarsMap }

-- merge s₂ into s₁
private def merge (s₁ s₂ : NameSet) : M NameSet :=
  s₂.foldM (init := s₁) fun s₁ k => do
    if s₁.contains k then
      pure s₁
    else
      markModified
      pure $ s₁.insert k

private def updateUsedVarsOf (fvarId : FVarId) : M Unit := do
  let usedFVarsMap ← getUsedFVarsMap
  match usedFVarsMap.find? fvarId with
  | none         => pure ()
  | some fvarIds =>
    let fvarIdsNew ← fvarIds.foldM (init := fvarIds) fun fvarIdsNew fvarId' =>
      if fvarId == fvarId' then
        pure fvarIdsNew
      else
        match usedFVarsMap.find? fvarId' with
        | none => pure fvarIdsNew
          /- We are being sloppy here `otherFVarIds` may contain free variables that are
             not in the context of the let-rec associated with fvarId.
````
````

```
             We filter these out-of-context free variables later. -/
        | some otherFVarIds => merge fvarIdsNew otherFVarIds
    modifyUsedFVars fun usedFVars => usedFVars.insert fvarId fvarIdsNew

private partial def fixpoint : Unit → M Unit
  | _ => do
    resetModified
    let letRecFVarIds ← read
    letRecFVarIds.forM updateUsedVarsOf
    if (← isModified) then
      fixpoint ()

def run (letRecFVarIds : List FVarId) (usedFVarsMap : UsedFVarsMap) : UsedFVarsMap :=
  let (_, s) := ((fixpoint ()).run letRecFVarIds).run { usedFVarsMap := usedFVarsMap }
  s.usedFVarsMap

end FixPoint

abbrev FreeVarMap := NameMap (Array FVarId)

private def mkFreeVarMap
    (mctx : MetavarContext) (sectionVars : Array Expr) (mainFVarIds : Array FVarId)
    (recFVarIds : Array FVarId) (letRecsToLift : List LetRecToLift) : FreeVarMap := do
  let usedFVarsMap  := mkInitialUsedFVarsMap mctx sectionVars mainFVarIds letRecsToLift
  let letRecFVarIds := letRecsToLift.map fun toLift => toLift.fvarId
  let usedFVarsMap  := FixPoint.run letRecFVarIds usedFVarsMap
  let mut freeVarMap := {}
  for toLift in letRecsToLift do
    let lctx       := toLift.lctx
    let fvarIdsSet := (usedFVarsMap.find? toLift.fvarId).get!
    let fvarIds    := fvarIdsSet.fold (init := #[]) fun fvarIds fvarId =>
      if lctx.contains fvarId && !recFVarIds.contains fvarId then
        fvarIds.push fvarId
      else
        fvarIds
    freeVarMap := freeVarMap.insert toLift.fvarId fvarIds
  pure freeVarMap

structure ClosureState where
  newLocalDecls : Array LocalDecl := #[]
  localDecls    : Array LocalDecl := #[]
  newLetDecls   : Array LocalDecl := #[]
  exprArgs      : Array Expr      := #[]

private def pickMaxFVar? (lctx : LocalContext) (fvarIds : Array FVarId) : Option FVarId :=
  fvarIds.getMax? fun fvarId₁ fvarId₂ => (lctx.get! fvarId₁).index < (lctx.get! fvarId₂).index

```

```
private def preprocess (e : Expr) : TermElabM Expr := do
  let e ← instantiateMVars e
  -- which let-decls are dependent. We say a let-decl is dependent if its lambda abstraction is type incorrect.
  Meta.check e
  pure e

/- Push free variables in `s` to `toProcess` if they are not already there. -/
private def pushNewVars (toProcess : Array FVarId) (s : CollectFVars.State) : Array FVarId :=
  s.fvarSet.fold (init := toProcess) fun toProcess fvarId =>
    if toProcess.contains fvarId then toProcess else toProcess.push fvarId

private def pushLocalDecl (toProcess : Array FVarId) (fvarId : FVarId) (userName : Name) (type : Expr) (bi := BinderInfo.default)
    : StateRefT ClosureState TermElabM (Array FVarId) := do
  let type ← preprocess type
  modify fun s => { s with
    newLocalDecls := s.newLocalDecls.push $ LocalDecl.cdecl arbitrary fvarId userName type bi,
    exprArgs      := s.exprArgs.push (mkFVar fvarId)
  }
  pure $ pushNewVars toProcess (collectFVars {} type)

private partial def mkClosureForAux (toProcess : Array FVarId) : StateRefT ClosureState TermElabM Unit := do
  let lctx ← getLCtx
  match pickMaxFVar? lctx toProcess with
  | none        => pure ()
  | some fvarId =>
    trace[Elab.definition.mkClosure] "toProcess: {toProcess.map mkFVar}, maxVar: {mkFVar fvarId}"
    let toProcess := toProcess.erase fvarId
    let localDecl ← getLocalDecl fvarId
    match localDecl with
    | LocalDecl.cdecl _ _ userName type bi =>
      let toProcess ← pushLocalDecl toProcess fvarId userName type bi
      mkClosureForAux toProcess
    | LocalDecl.ldecl _ _ userName type val _ =>
      let zetaFVarIds ← getZetaFVarIds
      if !zetaFVarIds.contains fvarId then
        /- Non-dependent let-decl. See comment at src/Lean/Meta/Closure.lean -/
        let toProcess ← pushLocalDecl toProcess fvarId userName type
        mkClosureForAux toProcess
      else
        /- Dependent let-decl. -/
        let type ← preprocess type
        let val  ← preprocess val
        modify fun s => { s with
          newLetDecls   := s.newLetDecls.push $ LocalDecl.ldecl arbitrary fvarId userName type val false,
          /- We don't want to interleave let and lambda declarations in our closure. So, we expand any occurrences of fvarId
             at `newLocalDecls` and `localDecls` -/
          newLocalDecls := s.newLocalDecls.map (replaceFVarIdAtLocalDecl fvarId val),
```

```
        localDecls := s.localDecls.map (replaceFVarIdAtLocalDecl fvarId val)
      }
      mkClosureForAux (pushNewVars toProcess (collectFVars (collectFVars {} type) val))

private partial def mkClosureFor (freeVars : Array FVarId) (localDecls : Array LocalDecl) : TermElabM ClosureState := do
  let (_, s) ← (mkClosureForAux freeVars).run { localDecls := localDecls }
  pure { s with
    newLocalDecls := s.newLocalDecls.reverse,
    newLetDecls   := s.newLetDecls.reverse,
    exprArgs      := s.exprArgs.reverse
  }

structure LetRecClosure where
  ref        : Syntax
  localDecls : Array LocalDecl
  closed     : Expr -- expression used to replace occurrences of the let-rec FVarId
  toLift     : LetRecToLift

private def mkLetRecClosureFor (toLift : LetRecToLift) (freeVars : Array FVarId) : TermElabM LetRecClosure := do
  let lctx := toLift.lctx
  withLCtx lctx toLift.localInstances do
  lambdaTelescope toLift.val fun xs val => do
    let type ← instantiateForall toLift.type xs
    let lctx ← getLCtx
    let s ← mkClosureFor freeVars $ xs.map fun x => lctx.get! x.fvarId!
    let type := Closure.mkForall s.localDecls $ Closure.mkForall s.newLetDecls type
    let val  := Closure.mkLambda s.localDecls $ Closure.mkLambda s.newLetDecls val
    let c    := mkAppN (Lean.mkConst toLift.declName) s.exprArgs
    assignExprMVar toLift.mvarId c
    return {
      ref        := toLift.ref
      localDecls := s.newLocalDecls
      closed     := c
      toLift     := { toLift with val := val, type := type }
    }

private def mkLetRecClosures (letRecsToLift : List LetRecToLift) (freeVarMap : FreeVarMap) : TermElabM (List LetRecClosure) :=
  letRecsToLift.mapM fun toLift => mkLetRecClosureFor toLift (freeVarMap.find? toLift.fvarId).get!

/- Mapping from FVarId of mutually recursive functions being defined to "closure" expression. -/
abbrev Replacement := NameMap Expr

def insertReplacementForMainFns (r : Replacement) (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainFVars : Array
  mainFVars.size.fold (init := r) fun i r =>
    r.insert mainFVars[i].fvarId! (mkAppN (Lean.mkConst mainHeaders[i].declName) sectionVars)
```

```
def insertReplacementForLetRecs (r : Replacement) (letRecClosures : List LetRecClosure) : Replacement :=
  letRecClosures.foldl (init := r) fun r c =>
    r.insert c.toLift.fvarId c.closed

def Replacement.apply (r : Replacement) (e : Expr) : Expr :=
  e.replace fun e => match e with
    | Expr.fvar fvarId _ => match r.find? fvarId with
      | some c => some c
      | _      => none
    | _ => none

def pushMain (preDefs : Array PreDefinition) (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainVals : Array Expr)
    : TermElabM (Array PreDefinition) :=
  mainHeaders.size.foldM (init := preDefs) fun i preDefs => do
    let header := mainHeaders[i]
    let val  ← mkLambdaFVars sectionVars mainVals[i]
    let type ← mkForallFVars sectionVars header.type
    return preDefs.push {
      ref         := getDeclarationSelectionRef header.ref
      kind        := header.kind
      declName    := header.declName
      levelParams := [], -- we set it later
      modifiers   := header.modifiers
      type        := type
      value       := val
    }

def pushLetRecs (preDefs : Array PreDefinition) (letRecClosures : List LetRecClosure) (kind : DefKind) (modifiers : Modifiers) : Array P
  letRecClosures.foldl (init := preDefs) fun preDefs c =>
    let type := Closure.mkForall c.localDecls c.toLift.type
    let val  := Closure.mkLambda c.localDecls c.toLift.val
    preDefs.push {
      ref         := c.ref
      kind        := kind
      declName    := c.toLift.declName
      levelParams := [] -- we set it later
      modifiers   := { modifiers with attrs := c.toLift.attrs }
      type        := type
      value       := val
    }

def getKindForLetRecs (mainHeaders : Array DefViewElabHeader) : DefKind :=
  if mainHeaders.any fun h => h.kind.isTheorem then DefKind.«theorem»
  else DefKind.«def»

def getModifiersForLetRecs (mainHeaders : Array DefViewElabHeader) : Modifiers := {
  isNoncomputable := mainHeaders.any fun h => h.modifiers.isNoncomputable,
```

```
  isPartial       := mainHeaders.any fun h => h.modifiers.isPartial,
  isUnsafe        := mainHeaders.any fun h => h.modifiers.isUnsafe
}

/-
- `sectionVars`:   The section variables used in the `mutual` block.
- `mainHeaders`:   The elaborated header of the top-level definitions being defined by the mutual block.
- `mainFVars`:     The auxiliary variables used to represent the top-level definitions being defined by the mutual block.
- `mainVals`:      The elaborated value for the top-level definitions
- `letRecsToLift`: The let-rec's definitions that need to be lifted
-/
def main (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainFVars : Array Expr) (mainVals : Array Expr) (letRecsToL
    : TermElabM (Array PreDefinition) := do
  -- Store in recFVarIds the fvarId of every function being defined by the mutual block.
  let mainFVarIds := mainFVars.map Expr.fvarId!
  let recFVarIds  := (letRecsToLift.toArray.map fun toLift => toLift.fvarId) ++ mainFVarIds
  -- Compute the set of free variables (excluding `recFVarIds`) for each let-rec.
  let mctx ← getMCtx
  let freeVarMap := mkFreeVarMap mctx sectionVars mainFVarIds recFVarIds letRecsToLift
  resetZetaFVarIds
  withTrackingZeta do
    -- By checking `toLift.type` and `toLift.val` we populate `zetaFVarIds`. See comments at `src/Lean/Meta/Closure.lean`.
    letRecsToLift.forM fun toLift => withLCtx toLift.lctx toLift.localInstances do Meta.check toLift.type; Meta.check toLift.val
    let letRecClosures ← mkLetRecClosures letRecsToLift freeVarMap
    -- mkLetRecClosures assign metavariables that were placeholders for the lifted declarations.
    let mainVals    ← mainVals.mapM (instantiateMVars ·)
    let mainHeaders ← mainHeaders.mapM instantiateMVarsAtHeader
    let letRecClosures ← letRecClosures.mapM fun closure => do pure { closure with toLift := (← instantiateMVarsAtLetRecToLift closure.t
    -- Replace fvarIds for functions being defined with closed terms
    let r              := insertReplacementForMainFns {} sectionVars mainHeaders mainFVars
    let r              := insertReplacementForLetRecs r letRecClosures
    let mainVals       := mainVals.map r.apply
    let mainHeaders    := mainHeaders.map fun h => { h with type := r.apply h.type }
    let letRecClosures := letRecClosures.map fun c => { c with toLift := { c.toLift with type := r.apply c.toLift.type, val := r.apply c
    let letRecKind     := getKindForLetRecs mainHeaders
    let letRecMods     := getModifiersForLetRecs mainHeaders
    pushMain (pushLetRecs #[] letRecClosures letRecKind letRecMods) sectionVars mainHeaders mainVals

end MutualClosure

private def getAllUserLevelNames (headers : Array DefViewElabHeader) : List Name :=
  if h : 0 < headers.size then
    -- Recall that all top-level functions must have the same levels. See `check` method above
    (headers.get ⟨0, h⟩).levelNames
  else
    []

```

```
/-- Eagerly convert universe metavariables occurring in theorem headers to universe parameters. -/
private def levelMVarToParamHeaders (views : Array DefView) (headers : Array DefViewElabHeader) : TermElabM (Array DefViewElabHeader) :=
  let rec process : StateRefT Nat TermElabM (Array DefViewElabHeader) := do
    let mut newHeaders := #[]
    for view in views, header in headers do
      if view.kind.isTheorem then
        newHeaders := newHeaders.push { header with type := (← levelMVarToParam' header.type) }
      else
        newHeaders := newHeaders.push header
    return newHeaders
  let newHeaders ← process.run' 1
  newHeaders.mapM fun header => return { header with type := (← instantiateMVars header.type) }

def elabMutualDef (vars : Array Expr) (views : Array DefView) : TermElabM Unit := do
  let scopeLevelNames ← getLevelNames
  let headers ← elabHeaders views
  let headers ← levelMVarToParamHeaders views headers
  let allUserLevelNames := getAllUserLevelNames headers
  withFunLocalDecls headers fun funFVars => do
    let values ← elabFunValues headers
    Term.synthesizeSyntheticMVarsNoPostponing
    let values ← values.mapM (instantiateMVars ·)
    let headers ← headers.mapM instantiateMVarsAtHeader
    let letRecsToLift ← getLetRecsToLift
    let letRecsToLift ← letRecsToLift.mapM instantiateMVarsAtLetRecToLift
    checkLetRecsToLiftTypes funFVars letRecsToLift
    withUsed vars headers values letRecsToLift fun vars => do
      let preDefs ← MutualClosure.main vars headers funFVars values letRecsToLift
      let preDefs ← levelMVarToParamPreDecls preDefs
      let preDefs ← instantiateMVarsAtPreDecls preDefs
      let preDefs ← fixLevelParams preDefs scopeLevelNames allUserLevelNames
      if isExample views then
        withoutModifyingEnv <| addPreDefinitions preDefs
      else
        addPreDefinitions preDefs

end Term
namespace Command

def elabMutualDef (ds : Array Syntax) : CommandElabM Unit := do
  let views ← ds.mapM fun d => do
    let modifiers ← elabModifiers d[0]
    mkDefView modifiers d[1]
  runTermElabM none fun vars => Term.elabMutualDef vars views

end Command
end Lean.Elab
```

```
::::::::::::::
Elab/Open.lean
::::::::::::::
/-
Copyright (c) 2021 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Log

namespace Lean.Elab
namespace OpenDecl

variable [Monad m] [STWorld IO.RealWorld m] [MonadEnv m]
variable [MonadExceptOf Exception m] [MonadRef m] [AddErrorMessageContext m]
variable [AddMessageContext m] [MonadLiftT (ST IO.RealWorld) m] [MonadLog m]

structure State where
  openDecls     : List OpenDecl
  currNamespace : Name

abbrev M := StateRefT State m

instance : MonadResolveName (M (m := m)) where
  getCurrNamespace   := return (← get).currNamespace
  getOpenDecls       := return (← get).openDecls

private def addOpenDecl (decl : OpenDecl) : M (m:=m) Unit :=
  modify fun s => { s with openDecls := decl :: s.openDecls }

private def elabOpenSimple (n : Syntax) : M (m:=m) Unit :=
  -- `open` id+
  for ns in n[0].getArgs do
    let ns ← resolveNamespace ns.getId
    addOpenDecl (OpenDecl.simple ns [])
    activateScoped ns

-- `open` id `(` id+ `)`
private def elabOpenOnly (n : Syntax) : M (m:=m) Unit := do
  let ns ← resolveNamespace n[0].getId
  for idStx in n[2].getArgs do
    let id := idStx.getId
    let declName := ns ++ id
    if (← getEnv).contains declName then
      addOpenDecl (OpenDecl.explicit id declName)
    else
      withRef idStx <| logUnknownDecl declName
```

```
-- `open` id `hiding` id+
private def elabOpenHiding (n : Syntax) : M (m:=m) Unit := do
  let ns ← resolveNamespace n[0].getId
  let mut ids : List Name := []
  for idStx in n[2].getArgs do
    let id := idStx.getId
    let declName := ns ++ id
    if (← getEnv).contains declName then
      ids := id::ids
    else
      withRef idStx <| logUnknownDecl declName
  addOpenDecl (OpenDecl.simple ns ids)

-- `open` id `renaming` sepBy (id `->` id) `,`
private def elabOpenRenaming (n : Syntax) : M (m:=m) Unit := do
  let ns ← resolveNamespace n[0].getId
  for stx in n[2].getSepArgs do
    let fromId   := stx[0].getId
    let toId     := stx[2].getId
    let declName := ns ++ fromId
    if (← getEnv).contains declName then
      addOpenDecl (OpenDecl.explicit toId declName)
    else
      withRef stx do logUnknownDecl declName

def elabOpenDecl [MonadResolveName m] (openDeclStx : Syntax) : m (List OpenDecl) := do
  StateRefT'.run' (s := { openDecls := (← getOpenDecls), currNamespace := (← getCurrNamespace) }) do
    if openDeclStx.getKind == ``Parser.Command.openSimple then
      elabOpenSimple openDeclStx
    else if openDeclStx.getKind == ``Parser.Command.openOnly then
      elabOpenOnly openDeclStx
    else if openDeclStx.getKind == ``Parser.Command.openHiding then
      elabOpenHiding openDeclStx
    else
      elabOpenRenaming openDeclStx
    return (← get).openDecls

end OpenDecl

export OpenDecl (elabOpenDecl)

end Lean.Elab::::::::::::::
Elab/PreDefinition.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.PreDefinition.Basic
import Lean.Elab.PreDefinition.Structural
import Lean.Elab.PreDefinition.Main
import Lean.Elab.PreDefinition.MkInhabitant
::::::::::::::
Elab/Print.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.FoldConsts
import Lean.Elab.Command

namespace Lean.Elab.Command

private def throwUnknownId (id : Name) : CommandElabM Unit :=
  throwError "unknown identifier '{mkConst id}'"

private def levelParamsToMessageData (levelParams : List Name) : MessageData :=
  match levelParams with
  | []    => ""
  | u::us => do
    let mut m := m!".\{{u}"
    for u in us do
      m := m ++ ", " ++ u
    return m ++ "}"

private def mkHeader (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (safety : DefinitionSafety) : CommandElabM Mess
  let m : MessageData :=
    match safety with
    | DefinitionSafety.unsafe  => "unsafe "
    | DefinitionSafety.partial => "partial "
    | DefinitionSafety.safe    => ""
  let m := if isProtected (← getEnv) id then m ++ "protected " else m
  let (m, id) := match privateToUserName? id with
    | some id => (m ++ "private ", id)
    | none    => (m, id)
  let m := m ++ kind ++ " " ++ id ++ levelParamsToMessageData levelParams ++ " : " ++ type
  pure m

private def mkHeader' (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (isUnsafe : Bool) : CommandElabM MessageData :
  mkHeader kind id levelParams type (if isUnsafe then DefinitionSafety.unsafe else DefinitionSafety.safe)
```

```
private def printDefLike (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (value : Expr) (safety := DefinitionSafety.
  let m ← mkHeader kind id levelParams type safety
  let m := m ++ " :=" ++ Format.line ++ value
  logInfo m

private def printAxiomLike (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (isUnsafe := false) : CommandElabM Unit :
  logInfo (← mkHeader' kind id levelParams type isUnsafe)

private def printQuot (kind : QuotKind) (id : Name) (levelParams : List Name) (type : Expr) : CommandElabM Unit := do
  printAxiomLike "Quotient primitive" id levelParams type

private def printInduct (id : Name) (levelParams : List Name) (numParams : Nat) (numIndices : Nat) (type : Expr)
    (ctors : List Name) (isUnsafe : Bool) : CommandElabM Unit := do
  let mut m ← mkHeader' "inductive" id levelParams type isUnsafe
  m := m ++ Format.line ++ "constructors:"
  for ctor in ctors do
    let cinfo ← getConstInfo ctor
    m := m ++ Format.line ++ ctor ++ " : " ++ cinfo.type
  logInfo m

private def printIdCore (id : Name) : CommandElabM Unit := do
  match (← getEnv).find? id with
  | ConstantInfo.axiomInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "axiom" id us t u
  | ConstantInfo.defnInfo  { levelParams := us, type := t, value := v, safety := s, .. } => printDefLike "def" id us t v s
  | ConstantInfo.thmInfo  { levelParams := us, type := t, value := v, .. } => printDefLike "theorem" id us t v
  | ConstantInfo.opaqueInfo  { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "constant" id us t u
  | ConstantInfo.quotInfo  { kind := kind, levelParams := us, type := t, .. } => printQuot kind id us t
  | ConstantInfo.ctorInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "constructor" id us t u
  | ConstantInfo.recInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "recursor" id us t u
  | ConstantInfo.inductInfo { levelParams := us, numParams := numParams, numIndices := numIndices, type := t, ctors := ctors, isUnsafe :
    printInduct id us numParams numIndices t ctors u
  | none => throwUnknownId id

private def printId (id : Syntax) : CommandElabM Unit := do
  let cs ← resolveGlobalConstWithInfos id
  cs.forM printIdCore

@[builtinCommandElab «print»] def elabPrint : CommandElab
  | `(#print%$tk $id:ident) => withRef tk <| printId id
  | `(#print%$tk $s:strLit) => logInfoAt tk s.isStrLit?.get!
  | _                       => throwError "invalid #print command"

namespace CollectAxioms

structure State where
  visited : NameSet    := {}
```

```
  axioms  : Array Name := #[]

abbrev M := ReaderT Environment $ StateM State

partial def collect (c : Name) : M Unit := do
  let collectExpr (e : Expr) : M Unit := e.getUsedConstants.forM collect
  let s ← get
  unless s.visited.contains c do
    modify fun s => { s with visited := s.visited.insert c }
    let env ← read
    match env.find? c with
    | some (ConstantInfo.axiomInfo _)  => modify fun s => { s with axioms := s.axioms.push c }
    | some (ConstantInfo.defnInfo v)   => collectExpr v.type *> collectExpr v.value
    | some (ConstantInfo.thmInfo v)    => collectExpr v.type *> collectExpr v.value
    | some (ConstantInfo.opaqueInfo v) => collectExpr v.type *> collectExpr v.value
    | some (ConstantInfo.quotInfo _)   => pure ()
    | some (ConstantInfo.ctorInfo v)   => collectExpr v.type
    | some (ConstantInfo.recInfo v)    => collectExpr v.type
    | some (ConstantInfo.inductInfo v) => collectExpr v.type *> v.ctors.forM collect
    | none                             => pure ()

end CollectAxioms

private def printAxiomsOf (constName : Name) : CommandElabM Unit := do
  let env ← getEnv
  let (_, s) := ((CollectAxioms.collect constName).run env).run {}
  if s.axioms.isEmpty then
    logInfo m!"'{constName}' does not depend on any axioms"
  else
    logInfo m!"'{constName}' depends on axioms: {s.axioms.toList}"

@[builtinCommandElab «printAxioms»] def elabPrintAxioms : CommandElab
  | `(#print%$tk axioms $id) => withRef tk do
    let cs ← resolveGlobalConstWithInfos id
    cs.forM printAxiomsOf
  | _ => throwUnsupportedSyntax

end Lean.Elab.Command
::::::::::::::
Elab/Quotation.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Sebastian Ullrich

Elaboration of syntax quotations as terms and patterns (in `match_syntax`). See also `./Hygiene.lean` for the basic
```

```
hygiene workings and data types.
-/
import Lean.Syntax
import Lean.ResolveName
import Lean.Elab.Term
import Lean.Elab.Quotation.Util
import Lean.Parser.Term

namespace Lean.Elab.Term.Quotation
open Lean.Parser.Term
open Lean.Syntax
open Meta

register_builtin_option hygiene : Bool := {
  defValue := true
  descr    := "Annotate identifiers in quotations such that they are resolved relative to the scope at their declaration, not that at th
}

/-- `C[$(e)]` ~> `let a := e; C[$a]`. Used in the implementation of antiquot splices. -/
private partial def floatOutAntiquotTerms : Syntax → StateT (Syntax → TermElabM Syntax) TermElabM Syntax
  | stx@(Syntax.node k args) => do
    if isAntiquot stx && !isEscapedAntiquot stx then
      let e := getAntiquotTerm stx
      if !e.isIdent || !e.getId.isAtomic then
        return ← withFreshMacroScope do
          let a ← `(a)
          modify (fun cont stx => (`(let $a:ident := $e; $stx) : TermElabM _))
          stx.setArg 2 a
    Syntax.node k (← args.mapM floatOutAntiquotTerms)
  | stx => pure stx

private def getSepFromSplice (splice : Syntax) : Syntax := do
  let Syntax.atom _ sep ← getAntiquotSpliceSuffix splice | unreachable!
  Syntax.mkStrLit (sep.dropRight 1)

partial def mkTuple : Array Syntax → TermElabM Syntax
  | #[]  => `(Unit.unit)
  | #[e] => e
  | es   => do
    let stx ← mkTuple (es.eraseIdx 0)
    `(Prod.mk $(es[0]) $stx)

def resolveSectionVariable (sectionVars : NameMap Name) (id : Name) : List (Name × List String) :=
  -- decode macro scopes from name before recursion
  let extractionResult := extractMacroScopes id
  let rec loop : Name → List String → List (Name × List String)
    | id@(Name.str p s _), projs =>
```

```
      -- NOTE: we assume that macro scopes always belong to the projected constant, not the projections
      let id := { extractionResult with name := id }.review
      match sectionVars.find? id with
      | some newId => [(newId, projs)]
      | none       => loop p (s::projs)
    | _, _ => []
  loop extractionResult.name []

-- Elaborate the content of a syntax quotation term
private partial def quoteSyntax : Syntax → TermElabM Syntax
  | Syntax.ident info rawVal val preresolved => do
    if !hygiene.get (← getOptions) then
      return ← `(Syntax.ident info $(quote rawVal) $(quote val) $(quote preresolved))
    -- Add global scopes at compilation time (now), add macro scope at runtime (in the quotation).
    -- See the paper for details.
    let r ← resolveGlobalName val
    -- extension of the paper algorithm: also store unique section variable names as top-level scopes
    -- so they can be captured and used inside the section, but not outside
    let r' := resolveSectionVariable (← read).sectionVars val
    let preresolved := r ++ r' ++ preresolved
    let val := quote val
    -- `scp` is bound in stxQuot.expand
    `(Syntax.ident info $(quote rawVal) (addMacroScope mainModule $val scp) $(quote preresolved))
  -- if antiquotation, insert contents as-is, else recurse
  | stx@(Syntax.node k _) => do
    if isAntiquot stx && !isEscapedAntiquot stx then
      getAntiquotTerm stx
    else if isTokenAntiquot stx && !isEscapedAntiquot stx then
      match stx[0] with
      | Syntax.atom _ val => `(Syntax.atom (Option.getD (getHeadInfo $(getAntiquotTerm stx)) info) $(quote val))
      | _                 => throwErrorAt stx "expected token"
    else if isAntiquotSuffixSplice stx && !isEscapedAntiquot stx then
      -- splices must occur in a `many` node
      throwErrorAt stx "unexpected antiquotation splice"
    else if isAntiquotSplice stx && !isEscapedAntiquot stx then
      throwErrorAt stx "unexpected antiquotation splice"
    else
      let empty ← `(Array.empty);
      -- if escaped antiquotation, decrement by one escape level
      let stx := unescapeAntiquot stx
      let args ← stx.getArgs.foldlM (fun args arg => do
        if k == nullKind && isAntiquotSuffixSplice arg then
          let antiquot := getAntiquotSuffixSpliceInner arg
          match antiquotSuffixSplice? arg with
          | `optional => `(Array.appendCore $args (match $(getAntiquotTerm antiquot):term with
            | some x => Array.empty.push x
            | none   => Array.empty))
```

```
        | `many     => `(Array.appendCore $args $(getAntiquotTerm antiquot))
        | `sepBy    => `(Array.appendCore $args (@SepArray.elemsAndSeps $(getSepFromSplice arg) $(getAntiquotTerm antiquot)))
        | k         => throwErrorAt arg "invalid antiquotation suffix splice kind '{k}'"
      else if k == nullKind && isAntiquotSplice arg then
        let k := antiquotSpliceKind? arg
        let (arg, bindLets) ← floatOutAntiquotTerms arg |>.run pure
        let inner ← (getAntiquotSpliceContents arg).mapM quoteSyntax
        let ids ← getAntiquotationIds arg
        if ids.isEmpty then
          throwErrorAt stx "antiquotation splice must contain at least one antiquotation"
        let arr ← match k with
          | `optional => `(match $[$ids:ident],* with
              | $[some $ids:ident],* => $(quote inner)
              | none                 => Array.empty)
          | _ =>
            let arr ← ids[:ids.size-1].foldrM (fun id arr => `(Array.zip $id $arr)) ids.back
            `(Array.map (fun $(← mkTuple ids) => $(inner[0])) $arr)
        let arr ←
          if k == `sepBy then
            `(mkSepArray $arr (mkAtom $(getSepFromSplice arg)))
          else arr
        let arr ← bindLets arr
        `(Array.appendCore $args $arr)
      else do
        let arg ← quoteSyntax arg;
        `(Array.push $args $arg)) empty
    `(Syntax.node $(quote k) $args)
  | Syntax.atom _ val =>
    `(Syntax.atom info $(quote val))
  | Syntax.missing => throwUnsupportedSyntax

def stxQuot.expand (stx : Syntax) : TermElabM Syntax := do
  /- Syntax quotations are monadic values depending on the current macro scope. For efficiency, we bind
     the macro scope once for each quotation, then build the syntax tree in a completely pure computation
     depending on this binding. Note that regular function calls do not introduce a new macro scope (i.e.
     we preserve referential transparency), so we can refer to this same `scp` inside `quoteSyntax` by
     including it literally in a syntax quotation. -/
  -- TODO: simplify to `(do scp ← getCurrMacroScope; pure $(quoteSyntax quoted))
  let stx ← quoteSyntax stx.getQuotContent;
  `(Bind.bind MonadRef.mkInfoFromRefPos (fun info =>
      Bind.bind getCurrMacroScope (fun scp =>
        Bind.bind getMainModule (fun mainModule => Pure.pure $stx))))
  /- NOTE: It may seem like the newly introduced binding `scp` may accidentally
     capture identifiers in an antiquotation introduced by `quoteSyntax`. However,
     note that the syntax quotation above enjoys the same hygiene guarantees as
     anywhere else in Lean; that is, we implement hygienic quotations by making
     use of the hygienic quotation support of the bootstrapped Lean compiler!
```

```

    Aside: While this might sound "dangerous", it is in fact less reliant on a
    "chain of trust" than other bootstrapping parts of Lean: because this
    implementation itself never uses `scp` (or any other identifier) both inside
    and outside quotations, it can actually correctly be compiled by an
    unhygienic (but otherwise correct) implementation of syntax quotations. As
    long as it is then compiled again with the resulting executable (i.e. up to
    stage 2), the result is a correct hygienic implementation. In this sense the
    implementation is "self-stabilizing". It was in fact originally compiled
    by an unhygienic prototype implementation. -/

macro "elab_stx_quot" kind:ident : command =>
  `(@[builtinTermElab $kind:ident] def elabQuot : TermElab := adaptExpander stxQuot.expand)

--

elab_stx_quot Parser.Level.quot
elab_stx_quot Parser.Term.quot
elab_stx_quot Parser.Term.funBinder.quot
elab_stx_quot Parser.Term.bracketedBinder.quot
elab_stx_quot Parser.Term.matchDiscr.quot
elab_stx_quot Parser.Tactic.quot
elab_stx_quot Parser.Tactic.quotSeq
elab_stx_quot Parser.Term.stx.quot
elab_stx_quot Parser.Term.prec.quot
elab_stx_quot Parser.Term.attr.quot
elab_stx_quot Parser.Term.prio.quot
elab_stx_quot Parser.Term.doElem.quot
elab_stx_quot Parser.Term.dynamicQuot

/- match -/

-- an "alternative" of patterns plus right-hand side
private abbrev Alt := List Syntax × Syntax

/--
  In a single match step, we match the first discriminant against the "head" of the first pattern of the first
  alternative. This datatype describes what kind of check this involves, which helps other patterns decide if
  they are covered by the same check and don't have to be checked again (see also `MatchResult`). -/
inductive HeadCheck where
  -- match step that always succeeds: _, x, `($x), ...
  | unconditional
  -- match step based on kind and, optionally, arity of discriminant
  -- If `arity` is given, that number of new discriminants is introduced. `covered` patterns should then introduce the
  -- same number of new patterns.
  -- We actually check the arity at run time only in the case of `null` nodes since it should otherwise by implied by
  -- the node kind.
```

```
  -- without arity: `($x:k)
  -- with arity: any quotation without an antiquotation head pattern
  | shape (k : SyntaxNodeKind) (arity : Option Nat)
  -- Match step that succeeds on `null` nodes of arity at least `numPrefix + numSuffix`, introducing discriminants
  -- for the first `numPrefix` children, one `null` node for those in between, and for the `numSuffix` last children.
  -- example: `([$x, $xs,*, $y]) is `slice 2 2`
  | slice (numPrefix numSuffix : Nat)
  -- other, complicated match step that will probably only cover identical patterns
  -- example: antiquotation splices `($[...]*)
  | other (pat : Syntax)

open HeadCheck

/-- Describe whether a pattern is covered by a head check (induced by the pattern itself or a different pattern). -/
inductive MatchResult where
  -- Pattern agrees with head check, remove and transform remaining alternative.
  -- If `exhaustive` is `false`, *also* include unchanged alternative in the "no" branch.
  | covered (f : Alt → TermElabM Alt) (exhaustive : Bool)
  -- Pattern disagrees with head check, include in "no" branch only
  | uncovered
  -- Pattern is not quite sure yet; include unchanged in both branches
  | undecided

open MatchResult

/-- All necessary information on a pattern head. -/
structure HeadInfo where
  -- check induced by the pattern
  check : HeadCheck
  -- compute compatibility of pattern with given head check
  onMatch (taken : HeadCheck) : MatchResult
  -- actually run the specified head check, with the discriminant bound to `discr`
  doMatch (yes : (newDiscrs : List Syntax) → TermElabM Syntax) (no : TermElabM Syntax) : TermElabM Syntax

/-- Adapt alternatives that do not introduce new discriminants in `doMatch`, but are covered by those that do so. -/
private def noOpMatchAdaptPats : HeadCheck → Alt → Alt
  | shape k (some sz), (pats, rhs) => (List.replicate sz (Unhygienic.run `(_)) ++ pats, rhs)
  | slice p s,         (pats, rhs) => (List.replicate (p + 1 + s) (Unhygienic.run `(_)) ++ pats, rhs)
  | _,                 alt         => alt

private def adaptRhs (fn : Syntax → TermElabM Syntax) : Alt → TermElabM Alt
  | (pats, rhs) => do (pats, ← fn rhs)

private partial def getHeadInfo (alt : Alt) : TermElabM HeadInfo :=
  let pat := alt.fst.head!
  let unconditionally (rhsFn) := pure {
    check := unconditional,
```

```
  doMatch := fun yes no => yes [],
  onMatch := fun taken => covered (adaptRhs rhsFn ∘ noOpMatchAdaptPats taken) (match taken with | unconditional => true | _ => false)
}
-- quotation pattern
if isQuot pat then
  let quoted := getQuotContent pat
  if quoted.isAtom then
    -- We assume that atoms are uniquely determined by the node kind and never have to be checked
    unconditionally pure
  else if quoted.isTokenAntiquot then
    unconditionally (`(let $(quoted.getAntiquotTerm) := discr; $(·)))
  else if isAntiquot quoted && !isEscapedAntiquot quoted then
    -- quotation contains a single antiquotation
    let k := antiquotKind? quoted |>.get!
    match getAntiquotTerm quoted with
    | `(_) => unconditionally pure
    | `($id:ident) =>
      -- Antiquotation kinds like `$id:ident` influence the parser, but also need to be considered by
      -- `match` (but not by quotation terms). For example, `($id:ident) and `($e) are not
      -- distinguishable without checking the kind of the node to be captured. Note that some
      -- antiquotations like the latter one for terms do not correspond to any actual node kind
      -- (signified by `k == Name.anonymous`), so we would only check for `ident` here.
      --
      -- if stx.isOfKind `ident then
      --   let id := stx; let e := stx; ...
      -- else
      --   let e := stx; ...
      let rhsFn := (`(let $id := discr; $(·)))
      if k == Name.anonymous then unconditionally rhsFn else pure {
        check   := shape k none,
        onMatch := fun
          | taken@(shape k' sz) =>
            if k' == k then
              covered (adaptRhs rhsFn ∘ noOpMatchAdaptPats taken) (exhaustive := sz.isNone)
            else uncovered
          | _ => uncovered,
        doMatch := fun yes no => do `(cond (Syntax.isOfKind discr $(quote k)) $(← yes []) $(← no)),
      }
    | anti => throwErrorAt anti "unsupported antiquotation kind in pattern"
  else if isAntiquotSuffixSplice quoted then throwErrorAt quoted "unexpected antiquotation splice"
  else if isAntiquotSplice quoted then throwErrorAt quoted "unexpected antiquotation splice"
  else if quoted.getArgs.size == 1 && isAntiquotSuffixSplice quoted[0] then
    let anti := getAntiquotTerm (getAntiquotSuffixSpliceInner quoted[0])
    unconditionally fun rhs => match antiquotSuffixSplice? quoted[0] with
      | `optional => `(let $anti := Syntax.getOptional? discr; $rhs)
      | `many     => `(let $anti := Syntax.getArgs discr; $rhs)
      | `sepBy    => `(let $anti := @SepArray.mk $(getSepFromSplice quoted[0]) (Syntax.getArgs discr); $rhs)
```

```
        | k         => throwErrorAt quoted "invalid antiquotation suffix splice kind '{k}'"
    else if quoted.getArgs.size == 1 && isAntiquotSplice quoted[0] then pure {
      check   := other pat,
      onMatch := fun
        | other pat' => if pat' == pat then covered pure (exhaustive := true) else undecided
        | _          => undecided,
      doMatch := fun yes no => do
        let splice := quoted[0]
        let k := antiquotSpliceKind? splice
        let contents := getAntiquotSpliceContents splice
        let ids ← getAntiquotationIds splice
        let yes ← yes []
        let no ← no
        match k with
        | `optional =>
          let nones := mkArray ids.size (← `(none))
          `(let_delayed yes _ $ids* := $yes;
            if discr.isNone then yes () $[ $nones]*
            else match discr with
              | `($(mkNullNode contents)) => yes () $[ (some $ids)]*
              | _                         => $no)
        | _ =>
          let mut discrs ← `(Syntax.getArgs discr)
          if k == `sepBy then
            discrs ← `(Array.getSepElems $discrs)
          let tuple ← mkTuple ids
          let mut yes := yes
          let resId ← match ids with
            | #[id] => id
            | _     =>
              for id in ids do
                yes ← `(let $id := tuples.map (fun $tuple => $id); $yes)
              `(tuples)
          let contents := if contents.size == 1
            then contents[0]
            else mkNullNode contents
          `(match OptionM.run ($(discrs).sequenceMap fun
                | `($contents) => some $tuple
                | _            => none) with
              | some $resId => $yes
              | none => $no)
    }
    else if let some idx := quoted.getArgs.findIdx? (fun arg => isAntiquotSuffixSplice arg || isAntiquotSplice arg) then do
      /-
        pattern of the form `match discr, ... with | `(pat_0 ... pat_(idx-1) $[...]* pat_(idx+1) ...), ...`
        transform to
        ```
```
```

```
      if discr.getNumArgs >= $quoted.getNumArgs - 1 then
        match discr[0], ..., discr[idx-1], mkNullNode (discr.getArgs.extract idx (discr.getNumArgs - $numSuffix))), ..., discr[quoted.g
          | `(pat_0), ... `(pat_(idx-1)), `($[...])*, `(pat_(idx+1)), ...
      ```
    -/
    let numSuffix := quoted.getNumArgs - 1 - idx
    pure {
      check   := slice idx numSuffix
      onMatch := fun
        | slice p s =>
          if p == idx && s == numSuffix then
            let argPats := quoted.getArgs.mapIdx fun i arg =>
              let arg := if (i : Nat) == idx then mkNullNode #[arg] else arg
              Unhygienic.run `(`($(arg)))
            covered (fun (pats, rhs) => (argPats.toList ++ pats, rhs)) (exhaustive := true)
          else uncovered
        | _ => uncovered
      doMatch := fun yes no => do
        let prefixDiscrs ← (List.range idx).mapM (`(Syntax.getArg discr $(quote ·)))
        let sliceDiscr ← `(mkNullNode (discr.getArgs.extract $(quote idx) (discr.getNumArgs - $(quote numSuffix))))
        let suffixDiscrs ← (List.range numSuffix).mapM fun i =>
          `(Syntax.getArg discr (discr.getNumArgs - $(quote (numSuffix - i))))
        `(ite (GE.ge discr.getNumArgs $(quote (quoted.getNumArgs - 1)))
            $(← yes (prefixDiscrs ++ sliceDiscr :: suffixDiscrs))
            $(← no))
    }
  else
    -- not an antiquotation, or an escaped antiquotation: match head shape
    let quoted := unescapeAntiquot quoted
    let kind := quoted.getKind
    let argPats := quoted.getArgs.map fun arg => Unhygienic.run `(`($(arg)))
    pure {
      check := shape kind argPats.size,
      onMatch := fun taken =>
        if (match taken with | shape k' sz => k' == kind && sz == argPats.size | _ => false : Bool) then
          covered (fun (pats, rhs) => (argPats.toList ++ pats, rhs)) (exhaustive := true)
        else
          uncovered,
      doMatch := fun yes no => do
        let cond ← match kind with
        | `null => `(Syntax.matchesNull discr $(quote argPats.size))
        | `ident => `(Syntax.matchesIdent discr $(quote quoted.getId))
        | _   => `(Syntax.isOfKind discr $(quote kind))
        let newDiscrs ← (List.range argPats.size).mapM fun i => `(Syntax.getArg discr $(quote i))
        `(ite (Eq $cond true) $(← yes newDiscrs) $(← no))
    }
else match pat with
```
```

```
    | `(_)              => unconditionally pure
    | `($id:ident)      => unconditionally (`(let $id := discr; $(·)))
    | `($id:ident@$pat) => do
      let info ← getHeadInfo (pat::alt.1.tail!, alt.2)
      { info with onMatch := fun taken => match info.onMatch taken with
          | covered f exh => covered (fun alt => f alt >>= adaptRhs (`(let $id := discr; $(·)))) exh
          | r             => r }
    | _               => throwErrorAt pat "match_syntax: unexpected pattern kind {pat}"

-- Bind right-hand side to new `let_delayed` decl in order to prevent code duplication
private def deduplicate (floatedLetDecls : Array Syntax) : Alt → TermElabM (Array Syntax × Alt)
  -- NOTE: new macro scope so that introduced bindings do not collide
  | (pats, rhs) => do
    if let `($f:ident $[ $args:ident]*) := rhs then
      -- looks simple enough/created by this function, skip
      return (floatedLetDecls, (pats, rhs))
    withFreshMacroScope do
      match ← getPatternsVars pats.toArray with
      | #[] =>
        -- no antiquotations => introduce Unit parameter to preserve evaluation order
        let rhs' ← `(rhs Unit.unit)
        (floatedLetDecls.push (← `(letDecl|rhs _ := $rhs)), (pats, rhs'))
      | vars =>
        let rhs' ← `(rhs $vars*)
        (floatedLetDecls.push (← `(letDecl|rhs $vars:ident* := $rhs)), (pats, rhs'))

private partial def compileStxMatch (discrs : List Syntax) (alts : List Alt) : TermElabM Syntax := do
  trace[Elab.match_syntax] "match {discrs} with {alts}"
  match discrs, alts with
  | [],            ([], rhs)::_ => pure rhs  -- nothing left to match
  | _,             []           => throwError "non-exhaustive 'match_syntax'"
  | discr::discrs, alt::alts    => do
    let info ← getHeadInfo alt
    let pat  := alt.1.head!
    let alts ← (alt::alts).mapM fun alt => do ((← getHeadInfo alt).onMatch info.check, alt)
    let mut yesAlts           := #[]
    let mut undecidedAlts     := #[]
    let mut nonExhaustiveAlts := #[]
    let mut floatedLetDecls   := #[]
    for alt in alts do
      let mut alt := alt
      match alt with
      | (covered f exh, alt) =>
        -- we can only factor out a common check if there are no undecided patterns in between;
        -- otherwise we would change the order of alternatives
        if undecidedAlts.isEmpty then
          yesAlts ← yesAlts.push <$> f (alt.1.tail!, alt.2)
```

```
          if !exh then
            nonExhaustiveAlts := nonExhaustiveAlts.push alt
        else
          (floatedLetDecls, alt) ← deduplicate floatedLetDecls alt
          undecidedAlts := undecidedAlts.push alt
          nonExhaustiveAlts := nonExhaustiveAlts.push alt
      | (undecided, alt) =>
        (floatedLetDecls, alt) ← deduplicate floatedLetDecls alt
        undecidedAlts := undecidedAlts.push alt
        nonExhaustiveAlts := nonExhaustiveAlts.push alt
      | (uncovered, alt) =>
        nonExhaustiveAlts := nonExhaustiveAlts.push alt
    let mut stx ← info.doMatch
      (yes := fun newDiscrs => do
        let mut yesAlts := yesAlts
        if !undecidedAlts.isEmpty then
          -- group undecided alternatives in a new default case `| discr2, ... => match discr, discr2, ... with ...`
          let vars ← discrs.mapM fun _ => withFreshMacroScope `(discr)
          let pats := List.replicate newDiscrs.length (Unhygienic.run `(_)) ++ vars
          let alts ← undecidedAlts.mapM fun alt => `(matchAltExpr| | $(alt.1.toArray),* => $(alt.2))
          let rhs  ← `(match discr, $[$(vars.toArray):term],* with $alts:matchAlt*)
          yesAlts := yesAlts.push (pats, rhs)
        withFreshMacroScope $ compileStxMatch (newDiscrs ++ discrs) yesAlts.toList)
      (no := withFreshMacroScope $ compileStxMatch (discr::discrs) nonExhaustiveAlts.toList)
    for d in floatedLetDecls do
      stx ← `(let_delayed $d:letDecl; $stx)
    `(let discr := $discr; $stx)
  | _, _ => unreachable!

def match_syntax.expand (stx : Syntax) : TermElabM Syntax := do
  match stx with
  | `(match $[$discrs:term],* with $[| $[$patss],* => $rhss]*) => do
    if !patss.any (·.any (fun
      | `($id@$pat) => pat.isQuot
      | pat        => pat.isQuot)) then
      -- no quotations => fall back to regular `match`
      throwUnsupportedSyntax
    let stx ← compileStxMatch discrs.toList (patss.map (·.toList) |>.zip rhss).toList
    trace[Elab.match_syntax.result] "{stx}"
    stx
  | _ => throwUnsupportedSyntax

@[builtinTermElab «match»] def elabMatchSyntax : TermElab :=
  adaptExpander match_syntax.expand

builtin_initialize
  registerTraceClass `Elab.match_syntax
```

```
  registerTraceClass `Elab.match_syntax.result

end Lean.Elab.Term.Quotation
::::::::::::::
Elab/SetOption.lean
::::::::::::::
/-
Copyright (c) 2021 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Log
import Lean.Elab.InfoTree
namespace Lean.Elab

variable [Monad m] [MonadOptions m] [MonadExceptOf Exception m] [MonadRef m]
variable [AddErrorMessageContext m] [MonadLiftT (EIO Exception) m] [MonadInfoTree m]

def elabSetOption (id : Syntax) (val : Syntax) : m Options := do
  let optionName := id.getId.eraseMacroScopes
  match val.isStrLit? with
  | some str => setOption optionName (DataValue.ofString str)
  | none     =>
  match val.isNatLit? with
  | some num => setOption optionName (DataValue.ofNat num)
  | none     =>
  match val with
  | Syntax.atom _ "true"  => setOption optionName (DataValue.ofBool true)
  | Syntax.atom _ "false" => setOption optionName (DataValue.ofBool false)
  | _ =>
    addCompletionInfo <| CompletionInfo.option (← getRef)
    throwError "unexpected set_option value {val}"
where
  setOption (optionName : Name) (val : DataValue) : m Options := do
    let ref ← getRef
    let decl ← IO.toEIO (fun (ex : IO.Error) => Exception.error ref ex.toString) (getOptionDecl optionName)
    unless decl.defValue.sameCtor val do throwError "type mismatch at set_option"
    return (← getOptions).insert optionName val

end Lean.Elab
::::::::::::::
Elab/StructInst.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
```

```
-/
import Lean.Util.FindExpr
import Lean.Parser.Term
import Lean.Elab.App
import Lean.Elab.Binders

namespace Lean.Elab.Term.StructInst

open Std (HashMap)
open Meta

/-
  Structure instances are of the form:

      "{" >> optional (atomic (termParser >> " with "))
          >> manyIndent (group (structInstField >> optional ", "))
          >> optEllipsis
          >> optional (" : " >> termParser)
          >> " }"
-/

@[builtinMacro Lean.Parser.Term.structInst] def expandStructInstExpectedType : Macro := fun stx =>
  let expectedArg := stx[4]
  if expectedArg.isNone then
    Macro.throwUnsupported
  else
    let expected := expectedArg[1]
    let stxNew   := stx.setArg 4 mkNullNode
    `(($stxNew : $expected))

/-
If `stx` is of the form `{ s with ... }` and `s` is not a local variable, expand into `let src := s; { src with ... }`.

Note that this one is not a `Macro` because we need to access the local context.
-/
private def expandNonAtomicExplicitSource (stx : Syntax) : TermElabM (Option Syntax) :=
  withFreshMacroScope do
    let sourceOpt := stx[1]
    if sourceOpt.isNone then
      pure none
    else
      let source := sourceOpt[0]
      match (← isLocalIdent? source) with
      | some _ => pure none
      | none   =>
        if source.isMissing then
          throwAbortTerm
```

```
        else
          let src ← `(src)
          let sourceOpt := sourceOpt.setArg 0 src
          let stxNew    := stx.setArg 1 sourceOpt
          `(let src := $source; $stxNew)

inductive Source where
  | none     -- structure instance source has not been provieded
  | implicit (stx : Syntax) -- `..`
  | explicit (stx : Syntax) (src : Expr) -- `src with`
  deriving Inhabited

def Source.isNone : Source → Bool
  | Source.none => true
  | _           => false

def setStructSourceSyntax (structStx : Syntax) : Source → Syntax
  | Source.none           => (structStx.setArg 1 mkNullNode).setArg 3 mkNullNode
  | Source.implicit stx   => (structStx.setArg 1 mkNullNode).setArg 3 stx
  | Source.explicit stx _ => (structStx.setArg 1 stx).setArg 3 mkNullNode

private def getStructSource (stx : Syntax) : TermElabM Source :=
  withRef stx do
    let explicitSource := stx[1]
    let implicitSource := stx[3]
    if explicitSource.isNone && implicitSource[0].isNone then
      return Source.none
    else if explicitSource.isNone then
      return Source.implicit implicitSource
    else if implicitSource[0].isNone then
      let fvar? ← isLocalIdent? explicitSource[0]
      match fvar? with
      | none     => unreachable! -- expandNonAtomicExplicitSource must have been used when we get here
      | some src  => return Source.explicit explicitSource src
    else
      throwError "invalid structure instance `with` and `..` cannot be used together"

/-
  We say a `{ ... }` notation is a `modifyOp` if it contains only one
  ```
  def structInstArrayRef := leading_parser "[" >> termParser >>"]"
  ```
-/
private def isModifyOp? (stx : Syntax) : TermElabM (Option Syntax) := do
  let s? ← stx[2].getArgs.foldlM (init := none) fun s? p =>
    /- p is of the form `(group (structInstField >> optional ", "))` -/
    let arg := p[0]
```

```
    /- Remark: the syntax for `structInstField` is
       ```
       def structInstLVal   := leading_parser (ident <|> numLit <|> structInstArrayRef) >> many (group ("." >> (ident <|> numLit)) <|> s
       def structInstField := leading_parser structInstLVal >> " := " >> termParser
       ```
    -/
    let lval := arg[0]
    let k    := lval[0].getKind
    if k == `Lean.Parser.Term.structInstArrayRef then
      match s? with
      | none   => pure (some arg)
      | some s =>
        if s.getKind == `Lean.Parser.Term.structInstArrayRef then
          throwErrorAt arg "invalid \{...} notation, at most one `[..]` at a given level"
        else
          throwErrorAt arg "invalid \{...} notation, can't mix field and `[..]` at a given level"
    else
      match s? with
      | none   => pure (some arg)
      | some s =>
        if s.getKind == `Lean.Parser.Term.structInstArrayRef then
          throwErrorAt arg "invalid \{...} notation, can't mix field and `[..]` at a given level"
        else
          pure s?
  match s? with
  | none   => pure none
  | some s => if s[0][0].getKind == `Lean.Parser.Term.structInstArrayRef then pure s? else pure none

private def elabModifyOp (stx modifyOp source : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  let cont (val : Syntax) : TermElabM Expr := do
    let lval := modifyOp[0][0]
    let idx  := lval[1]
    let self := source[0]
    let stxNew ← `($(self).modifyOp (idx := $idx) (fun s => $val))
    trace[Elab.struct.modifyOp] "{stx}\n===>\n{stxNew}"
    withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
  trace[Elab.struct.modifyOp] "{modifyOp}\nSource: {source}"
  let rest := modifyOp[0][1]
  if rest.isNone then
    cont modifyOp[2]
  else
    let s ← `(s)
    let valFirst  := rest[0]
    let valFirst  := if valFirst.getKind == `Lean.Parser.Term.structInstArrayRef then valFirst else valFirst[1]
    let restArgs  := rest.getArgs
    let valRest   := mkNullNode restArgs[1:restArgs.size]
    let valField  := modifyOp.setArg 0 <| Syntax.node ``Parser.Term.structInstLVal #[valFirst, valRest]
```

```
    let valSource := source.modifyArg 0 fun _ => s
    let val       := stx.setArg 1 valSource
    let val       := val.setArg 2 <| mkNullNode #[mkNullNode #[valField, mkNullNode]]
    trace[Elab.struct.modifyOp] "{stx}\nval: {val}"
    cont val

/- Get structure name and elaborate explicit source (if available) -/
private def getStructName (stx : Syntax) (expectedType? : Option Expr) (sourceView : Source) : TermElabM (Name × Expr) := do
  tryPostponeIfNoneOrMVar expectedType?
  let useSource : Unit → TermElabM (Name × Expr) := fun _ =>
    match sourceView, expectedType? with
    | Source.explicit _ src, _ => do
      let srcType ← inferType src
      let srcType ← whnf srcType
      tryPostponeIfMVar srcType
      match srcType.getAppFn with
      | Expr.const constName _ _ => return (constName, srcType)
      | _ => throwUnexpectedExpectedType srcType "source"
    | _, some expectedType => throwUnexpectedExpectedType expectedType
    | _, none              => throwUnknownExpectedType
  match expectedType? with
  | none => useSource ()
  | some expectedType =>
    let expectedType ← whnf expectedType
    match expectedType.getAppFn with
    | Expr.const constName _ _ => return (constName, expectedType)
    | _                        => useSource ()
where
  throwUnknownExpectedType :=
    throwError "invalid \{...} notation, expected type is not known"
  throwUnexpectedExpectedType type (kind := "expected") := do
    let type ← instantiateMVars type
    if type.getAppFn.isMVar then
      throwUnknownExpectedType
    else
      throwError "invalid \{...} notation, {kind} type is not of the form (C ...){indentExpr type}"

inductive FieldLHS where
  | fieldName  (ref : Syntax) (name : Name)
  | fieldIndex (ref : Syntax) (idx : Nat)
  | modifyOp   (ref : Syntax) (index : Syntax)
  deriving Inhabited

instance : ToFormat FieldLHS := ⟨fun lhs =>
  match lhs with
  | FieldLHS.fieldName _ n  => fmt n
  | FieldLHS.fieldIndex _ i => fmt i
```

```
  | FieldLHS.modifyOp _ i   => "[" ++ i.prettyPrint ++ "]")

inductive FieldVal (σ : Type) where
  | term  (stx : Syntax) : FieldVal σ
  | nested (s : σ)       : FieldVal σ
  | default              : FieldVal σ -- mark that field must be synthesized using default value
  deriving Inhabited

structure Field (σ : Type) where
  ref : Syntax
  lhs : List FieldLHS
  val : FieldVal σ
  expr? : Option Expr := none
  deriving Inhabited

def Field.isSimple {σ} : Field σ → Bool
  | { lhs := [_], .. } => true
  | _                  => false

inductive Struct where
  | mk (ref : Syntax) (structName : Name) (fields : List (Field Struct)) (source : Source)
  deriving Inhabited

abbrev Fields := List (Field Struct)

/- true if all fields of the given structure are marked as `default` -/
partial def Struct.allDefault : Struct → Bool
  | ⟨_, _, fields, _⟩ => fields.all fun ⟨_, _, val, _⟩ => match val with
    | FieldVal.term _   => false
    | FieldVal.default  => true
    | FieldVal.nested s => allDefault s

def Struct.ref : Struct → Syntax
  | ⟨ref, _, _, _⟩ => ref

def Struct.structName : Struct → Name
  | ⟨_, structName, _, _⟩ => structName

def Struct.fields : Struct → Fields
  | ⟨_, _, fields, _⟩ => fields

def Struct.source : Struct → Source
  | ⟨_, _, _, s⟩ => s

def formatField (formatStruct : Struct → Format) (field : Field Struct) : Format :=
  Format.joinSep field.lhs " . " ++ " := " ++
    match field.val with
```

```
    | FieldVal.term v   => v.prettyPrint
    | FieldVal.nested s => formatStruct s
    | FieldVal.default  => "<default>"

partial def formatStruct : Struct → Format
  | ⟨_, structName, fields, source⟩ =>
    let fieldsFmt := Format.joinSep (fields.map (formatField formatStruct)) ", "
    match source with
    | Source.none           => "{" ++ fieldsFmt ++ "}"
    | Source.implicit _     => "{" ++ fieldsFmt ++ " .. }"
    | Source.explicit _ src => "{" ++ format src ++ " with " ++ fieldsFmt ++ "}"

instance : ToFormat Struct     := ⟨formatStruct⟩
instance : ToString Struct := ⟨toString ∘ format⟩

instance : ToFormat (Field Struct) := ⟨formatField formatStruct⟩
instance : ToString (Field Struct) := ⟨toString ∘ format⟩

/-
Recall that `structInstField` elements have the form
```
   def structInstField := leading_parser structInstLVal >> " := " >> termParser
   def structInstLVal   := leading_parser (ident <|> numLit <|> structInstArrayRef) >> many (("." >> (ident <|> numLit)) <|> structInstA
   def structInstArrayRef := leading_parser "[" >> termParser >>"]"
```
-/
-- Remark: this code relies on the fact that `expandStruct` only transforms `fieldLHS.fieldName`
def FieldLHS.toSyntax (first : Bool) : FieldLHS → Syntax
  | FieldLHS.modifyOp   stx _    => stx
  | FieldLHS.fieldName  stx name => if first then mkIdentFrom stx name else mkGroupNode #[mkAtomFrom stx ".", mkIdentFrom stx name]
  | FieldLHS.fieldIndex stx _    => if first then stx else mkGroupNode #[mkAtomFrom stx ".", stx]

def FieldVal.toSyntax : FieldVal Struct → Syntax
  | FieldVal.term stx => stx
  | _                 => unreachable!

def Field.toSyntax : Field Struct → Syntax
  | field =>
    let stx := field.ref
    let stx := stx.setArg 2 field.val.toSyntax
    match field.lhs with
    | first::rest => stx.setArg 0 <| mkNullNode #[first.toSyntax true, mkNullNode <| rest.toArray.map (FieldLHS.toSyntax false) ]
    | _ => unreachable!

private def toFieldLHS (stx : Syntax) : Except String FieldLHS :=
  if stx.getKind == `Lean.Parser.Term.structInstArrayRef then
    return FieldLHS.modifyOp stx stx[1]
```

```
  else
    -- Note that the representation of the first field is different.
    let stx := if stx.getKind == groupKind then stx[1] else stx
    if stx.isIdent then
      return FieldLHS.fieldName stx stx.getId.eraseMacroScopes
    else match stx.isFieldIdx? with
      | some idx => return FieldLHS.fieldIndex stx idx
      | none     => throw "unexpected structure syntax"

private def mkStructView (stx : Syntax) (structName : Name) (source : Source) : Except String Struct := do
  /- Recall that `stx` is of the form
     ```
     leading_parser "{" >> optional (atomic (termParser >> " with "))
                 >> manyIndent (group (structInstField >> optional ", "))
                 >> optional ".."
                 >> optional (" : " >> termParser)
                 >> " }"
     ```
  -/
  let fieldsStx := stx[2].getArgs.map (·[0])
  let fields ← fieldsStx.toList.mapM fun fieldStx => do
    let val   := fieldStx[2]
    let first ← toFieldLHS fieldStx[0][0]
    let rest  ← fieldStx[0][1].getArgs.toList.mapM toFieldLHS
    pure { ref := fieldStx, lhs := first :: rest, val := FieldVal.term val : Field Struct }
  pure ⟨stx, structName, fields, source⟩

def Struct.modifyFieldsM {m : Type → Type} [Monad m] (s : Struct) (f : Fields → m Fields) : m Struct :=
  match s with
  | ⟨ref, structName, fields, source⟩ => return ⟨ref, structName, (← f fields), source⟩

@[inline] def Struct.modifyFields (s : Struct) (f : Fields → Fields) : Struct :=
  Id.run <| s.modifyFieldsM f

def Struct.setFields (s : Struct) (fields : Fields) : Struct :=
  s.modifyFields fun _ => fields

private def expandCompositeFields (s : Struct) : Struct :=
  s.modifyFields fun fields => fields.map fun field => match field with
    | { lhs := FieldLHS.fieldName ref (Name.str Name.anonymous _ _) :: rest, .. } => field
    | { lhs := FieldLHS.fieldName ref n@(Name.str _ _ _) :: rest, .. } =>
      let newEntries := n.components.map <| FieldLHS.fieldName ref
      { field with lhs := newEntries ++ rest }
    | _ => field

private def expandNumLitFields (s : Struct) : TermElabM Struct :=
  s.modifyFieldsM fun fields => do
```

```
  let env ← getEnv
  let fieldNames := getStructureFields env s.structName
  fields.mapM fun field => match field with
    | { lhs := FieldLHS.fieldIndex ref idx :: rest, .. } =>
      if idx == 0 then throwErrorAt ref "invalid field index, index must be greater than 0"
      else if idx > fieldNames.size then throwErrorAt ref "invalid field index, structure has only #{fieldNames.size} fields"
      else pure { field with lhs := FieldLHS.fieldName ref fieldNames[idx - 1] :: rest }
    | _ => pure field

/- For example, consider the following structures:
   ```
   structure A where
     x : Nat

   structure B extends A where
     y : Nat

   structure C extends B where
     z : Bool
   ```
   This method expands parent structure fields using the path to the parent structure.
   For example,
   ```
   { x := 0, y := 0, z := true : C }
   ```
   is expanded into
   ```
   { toB.toA.x := 0, toB.y := 0, z := true : C }
   ```
-/
private def expandParentFields (s : Struct) : TermElabM Struct := do
  let env ← getEnv
  s.modifyFieldsM fun fields => fields.mapM fun field => match field with
    | { lhs := FieldLHS.fieldName ref fieldName :: rest, .. } =>
      match findField? env s.structName fieldName with
      | none => throwErrorAt ref "'{fieldName}' is not a field of structure '{s.structName}'"
      | some baseStructName =>
        if baseStructName == s.structName then pure field
        else match getPathToBaseStructure? env baseStructName s.structName with
          | some path => do
            let path := path.map fun funName => match funName with
              | Name.str _ s _ => FieldLHS.fieldName ref (Name.mkSimple s)
              | _              => unreachable!
            pure { field with lhs := path ++ field.lhs }
          | _ => throwErrorAt ref "failed to access field '{fieldName}' in parent structure"
    | _ => pure field
```

```
private abbrev FieldMap := HashMap Name Fields

private def mkFieldMap (fields : Fields) : TermElabM FieldMap :=
  fields.foldlM (init := {}) fun fieldMap field =>
    match field.lhs with
    | FieldLHS.fieldName _ fieldName :: rest =>
      match fieldMap.find? fieldName with
      | some (prevField::restFields) =>
        if field.isSimple || prevField.isSimple then
          throwErrorAt field.ref "field '{fieldName}' has already beed specified"
        else
          return fieldMap.insert fieldName (field::prevField::restFields)
      | _ => return fieldMap.insert fieldName [field]
    | _ => unreachable!

private def isSimpleField? : Fields → Option (Field Struct)
  | [field] => if field.isSimple then some field else none
  | _       => none

private def getFieldIdx (structName : Name) (fieldNames : Array Name) (fieldName : Name) : TermElabM Nat := do
  match fieldNames.findIdx? fun n => n == fieldName with
  | some idx => pure idx
  | none     => throwError "field '{fieldName}' is not a valid field of '{structName}'"

private def mkProjStx (s : Syntax) (fieldName : Name) : Syntax :=
  Syntax.node `Lean.Parser.Term.proj #[s, mkAtomFrom s ".", mkIdentFrom s fieldName]

private def mkSubstructSource (structName : Name) (fieldNames : Array Name) (fieldName : Name) (src : Source) : TermElabM Source :=
  match src with
  | Source.explicit stx src => do
    let idx ← getFieldIdx structName fieldNames fieldName
    let stx := stx.modifyArg 0 fun stx => mkProjStx stx fieldName
    return Source.explicit stx (mkProj structName idx src)
  | s => return s

@[specialize] private def groupFields (expandStruct : Struct → TermElabM Struct) (s : Struct) : TermElabM Struct := do
  let env ← getEnv
  let fieldNames := getStructureFields env s.structName
  withRef s.ref do
  s.modifyFieldsM fun fields => do
    let fieldMap ← mkFieldMap fields
    fieldMap.toList.mapM fun (fieldName, fields) => do
      match isSimpleField? fields with
      | some field => pure field
      | none =>
        let substructFields := fields.map fun field => { field with lhs := field.lhs.tail! }
        let substructSource ← mkSubstructSource s.structName fieldNames fieldName s.source
```

```
        let field := fields.head!
        match Lean.isSubobjectField? env s.structName fieldName with
        | some substructName =>
          let substruct := Struct.mk s.ref substructName substructFields substructSource
          let substruct ← expandStruct substruct
          pure { field with lhs := [field.lhs.head!], val := FieldVal.nested substruct }
        | none => do
          -- It is not a substructure field. Thus, we wrap fields using `Syntax`, and use `elabTerm` to process them.
          let valStx := s.ref -- construct substructure syntax using s.ref as template
          let valStx := valStx.setArg 4 mkNullNode -- erase optional expected type
          let args   := substructFields.toArray.map fun field => mkNullNode #[field.toSyntax, mkNullNode]
          let valStx := valStx.setArg 2 (mkNullNode args)
          let valStx := setStructSourceSyntax valStx substructSource
          pure { field with lhs := [field.lhs.head!], val := FieldVal.term valStx }

def findField? (fields : Fields) (fieldName : Name) : Option (Field Struct) :=
  fields.find? fun field =>
    match field.lhs with
    | [FieldLHS.fieldName _ n] => n == fieldName
    | _                        => false

@[specialize] private def addMissingFields (expandStruct : Struct → TermElabM Struct) (s : Struct) : TermElabM Struct := do
  let env ← getEnv
  let fieldNames := getStructureFields env s.structName
  let ref := s.ref
  withRef ref do
    let fields ← fieldNames.foldlM (init := []) fun fields fieldName => do
      match findField? s.fields fieldName with
      | some field => return field::fields
      | none       =>
        let addField (val : FieldVal Struct) : TermElabM Fields := do
          return { ref := s.ref, lhs := [FieldLHS.fieldName s.ref fieldName], val := val } :: fields
        match Lean.isSubobjectField? env s.structName fieldName with
        | some substructName => do
          let substructSource ← mkSubstructSource s.structName fieldNames fieldName s.source
          let substruct := Struct.mk s.ref substructName [] substructSource
          let substruct ← expandStruct substruct
          addField (FieldVal.nested substruct)
        | none =>
          match s.source with
          | Source.none           => addField FieldVal.default
          | Source.implicit _     => addField (FieldVal.term (mkHole s.ref))
          | Source.explicit stx _ =>
            -- stx is of the form `optional (try (termParser >> "with"))`
            let src := stx[0]
            let val := mkProjStx src fieldName
            addField (FieldVal.term val)
```

```
    return s.setFields fields.reverse

private partial def expandStruct (s : Struct) : TermElabM Struct := do
  let s := expandCompositeFields s
  let s ← expandNumLitFields s
  let s ← expandParentFields s
  let s ← groupFields expandStruct s
  addMissingFields expandStruct s

structure CtorHeaderResult where
  ctorFn     : Expr
  ctorFnType : Expr
  instMVars  : Array MVarId := #[]

private def mkCtorHeaderAux : Nat → Expr → Expr → Array MVarId → TermElabM CtorHeaderResult
  | 0,   type, ctorFn, instMVars => pure { ctorFn := ctorFn, ctorFnType := type, instMVars := instMVars }
  | n+1, type, ctorFn, instMVars => do
    let type ← whnfForall type
    match type with
    | Expr.forallE _ d b c =>
      match c.binderInfo with
      | BinderInfo.instImplicit =>
        let a ← mkFreshExprMVar d MetavarKind.synthetic
        mkCtorHeaderAux n (b.instantiate1 a) (mkApp ctorFn a) (instMVars.push a.mvarId!)
      | _ =>
        let a ← mkFreshExprMVar d
        mkCtorHeaderAux n (b.instantiate1 a) (mkApp ctorFn a) instMVars
    | _ => throwError "unexpected constructor type"

private partial def getForallBody : Nat → Expr → Option Expr
  | i+1, Expr.forallE _ _ b _ => getForallBody i b
  | i+1, _                    => none
  | 0,   type                 => type

private def propagateExpectedType (type : Expr) (numFields : Nat) (expectedType? : Option Expr) : TermElabM Unit :=
  match expectedType? with
  | none              => pure ()
  | some expectedType => do
    match getForallBody numFields type with
      | none           => pure ()
      | some typeBody =>
        unless typeBody.hasLooseBVars do
          discard <| isDefEq expectedType typeBody

private def mkCtorHeader (ctorVal : ConstructorVal) (expectedType? : Option Expr) : TermElabM CtorHeaderResult := do
  let us ← mkFreshLevelMVars ctorVal.levelParams.length
  let val  := Lean.mkConst ctorVal.name us
```

```
  let type := (ConstantInfo.ctorInfo ctorVal).instantiateTypeLevelParams us
  let r ← mkCtorHeaderAux ctorVal.numParams type val #[]
  propagateExpectedType r.ctorFnType ctorVal.numFields expectedType?
  synthesizeAppInstMVars r.instMVars
  pure r

def markDefaultMissing (e : Expr) : Expr :=
  mkAnnotation `structInstDefault e

def defaultMissing? (e : Expr) : Option Expr :=
  annotation? `structInstDefault e

def throwFailedToElabField {α} (fieldName : Name) (structName : Name) (msgData : MessageData) : TermElabM α :=
  throwError "failed to elaborate field '{fieldName}' of '{structName}, {msgData}"

def trySynthStructInstance? (s : Struct) (expectedType : Expr) : TermElabM (Option Expr) := do
  if !s.allDefault then
    pure none
  else
    try synthInstance? expectedType catch _ => pure none

private partial def elabStruct (s : Struct) (expectedType? : Option Expr) : TermElabM (Expr × Struct) := withRef s.ref do
  let env ← getEnv
  let ctorVal := getStructureCtor env s.structName
  let { ctorFn := ctorFn, ctorFnType := ctorFnType, .. } ← mkCtorHeader ctorVal expectedType?
  let (e, _, fields) ← s.fields.foldlM (init := (ctorFn, ctorFnType, [])) fun (e, type, fields) field =>
    match field.lhs with
    | [FieldLHS.fieldName ref fieldName] => do
      let type ← whnfForall type
      match type with
      | Expr.forallE _ d b c =>
        let cont (val : Expr) (field : Field Struct) : TermElabM (Expr × Expr × Fields) := do
          pushInfoTree <| InfoTree.node (children := {}) <| Info.ofFieldInfo { lctx := (← getLCtx), val := val, name := fieldName, stx :
          let e     := mkApp e val
          let type  := b.instantiate1 val
          let field := { field with expr? := some val }
          pure (e, type, field::fields)
        match field.val with
        | FieldVal.term stx => cont (← elabTermEnsuringType stx d) field
        | FieldVal.nested s => do
          -- if all fields of `s` are marked as `default`, then try to synthesize instance
          match (← trySynthStructInstance? s d) with
          | some val => cont val { field with val := FieldVal.term (mkHole field.ref) }
          | none     => do let (val, sNew) ← elabStruct s (some d); let val ← ensureHasType d val; cont val { field with val := FieldVal
        | FieldVal.default  => do let val ← withRef field.ref <| mkFreshExprMVar (some d); cont (markDefaultMissing val) field
      | _ => withRef field.ref <| throwFailedToElabField fieldName s.structName m!"unexpected constructor type{indentExpr type}"
    | _ => throwErrorAt field.ref "unexpected unexpanded structure field"
```

```
  pure (e, s.setFields fields.reverse)

namespace DefaultFields

structure Context where
  -- We must search for default values overriden in derived structures
  structs : Array Struct := #[]
  allStructNames : Array Name := #[]
  /--
  Consider the following example:
  ```
  structure A where
    x : Nat := 1

  structure B extends A where
    y : Nat := x + 1
    x := y + 1

  structure C extends B where
    z : Nat := 2*y
    x := z + 3
  ```
  And we are trying to elaborate a structure instance for `C`. There are default values for `x` at `A`, `B`, and `C`.
  We say the default value at `C` has distance 0, the one at `B` distance 1, and the one at `A` distance 2.
  The field `maxDistance` specifies the maximum distance considered in a round of Default field computation.
  Remark: since `C` does not set a default value of `y`, the default value at `B` is at distance 0.

  The fixpoint for setting default values works in the following way.
  - Keep computing default values using `maxDistance == 0`.
  - We increase `maxDistance` whenever we failed to compute a new default value in a round.
  - If `maxDistance > 0`, then we interrupt a round as soon as we compute some default value.
    We use depth-first search.
  - We sign an error if no progress is made when `maxDistance` == structure hierarchy depth (2 in the example above).
  -/
  maxDistance : Nat := 0

structure State where
  progress : Bool := false

partial def collectStructNames (struct : Struct) (names : Array Name) : Array Name :=
  let names := names.push struct.structName
  struct.fields.foldl (init := names) fun names field =>
    match field.val with
    | FieldVal.nested struct => collectStructNames struct names
    | _ => names

partial def getHierarchyDepth (struct : Struct) : Nat :=
```

```
  struct.fields.foldl (init := 0) fun max field =>
    match field.val with
    | FieldVal.nested struct => Nat.max max (getHierarchyDepth struct + 1)
    | _ => max

partial def findDefaultMissing? (mctx : MetavarContext) (struct : Struct) : Option (Field Struct) :=
  struct.fields.findSome? fun field =>
   match field.val with
   | FieldVal.nested struct => findDefaultMissing? mctx struct
   | _ => match field.expr? with
     | none      => unreachable!
     | some expr => match defaultMissing? expr with
       | some (Expr.mvar mvarId _) => if mctx.isExprAssigned mvarId then none else some field
       | _                         => none

def getFieldName (field : Field Struct) : Name :=
  match field.lhs with
  | [FieldLHS.fieldName _ fieldName] => fieldName
  | _ => unreachable!

abbrev M := ReaderT Context (StateRefT State TermElabM)

def isRoundDone : M Bool := do
  return (← get).progress && (← read).maxDistance > 0

def getFieldValue? (struct : Struct) (fieldName : Name) : Option Expr :=
  struct.fields.findSome? fun field =>
    if getFieldName field == fieldName then
      field.expr?
    else
      none

partial def mkDefaultValueAux? (struct : Struct) : Expr → TermElabM (Option Expr)
  | Expr.lam n d b c => withRef struct.ref do
    if c.binderInfo.isExplicit then
      let fieldName := n
      match getFieldValue? struct fieldName with
      | none     => pure none
      | some val =>
        let valType ← inferType val
        if (← isDefEq valType d) then
          mkDefaultValueAux? struct (b.instantiate1 val)
        else
          pure none
    else
      let arg ← mkFreshExprMVar d
      mkDefaultValueAux? struct (b.instantiate1 arg)
```

```
  | e =>
    if e.isAppOfArity `id 2 then
      pure (some e.appArg!)
    else
      pure (some e)

def mkDefaultValue? (struct : Struct) (cinfo : ConstantInfo) : TermElabM (Option Expr) :=
  withRef struct.ref do
  let us ← mkFreshLevelMVarsFor cinfo
  mkDefaultValueAux? struct (cinfo.instantiateValueLevelParams us)

/-- If `e` is a projection function of one of the given structures, then reduce it -/
def reduceProjOf? (structNames : Array Name) (e : Expr) : MetaM (Option Expr) := do
  if !e.isApp then pure none
  else match e.getAppFn with
    | Expr.const name _ _ => do
      let env ← getEnv
      match env.getProjectionStructureName? name with
      | some structName =>
        if structNames.contains structName then
          Meta.unfoldDefinition? e
        else
          pure none
      | none => pure none
    | _ => pure none

/-- Reduce default value. It performs beta reduction and projections of the given structures. -/
partial def reduce (structNames : Array Name) : Expr → MetaM Expr
  | e@(Expr.lam _ _ _ _)     => lambdaLetTelescope e fun xs b => do mkLambdaFVars xs (← reduce structNames b)
  | e@(Expr.forallE _ _ _ _) => forallTelescope e fun xs b => do mkForallFVars xs (← reduce structNames b)
  | e@(Expr.letE _ _ _ _ _)  => lambdaLetTelescope e fun xs b => do mkLetFVars xs (← reduce structNames b)
  | e@(Expr.proj _ i b _)    => do
    match (← Meta.project? b i) with
    | some r => reduce structNames r
    | none   => return e.updateProj! (← reduce structNames b)
  | e@(Expr.app f _ _) => do
    match (← reduceProjOf? structNames e) with
    | some r => reduce structNames r
    | none   =>
      let f := f.getAppFn
      let f' ← reduce structNames f
      if f'.isLambda then
        let revArgs := e.getAppRevArgs
        reduce structNames (f'.betaRev revArgs)
      else
        let args ← e.getAppArgs.mapM (reduce structNames)
        return (mkAppN f' args)
```

```
  | e@(Expr.mdata _ b _) => do
    let b ← reduce structNames b
    if (defaultMissing? e).isSome && !b.isMVar then
      return b
    else
      return e.updateMData! b
  | e@(Expr.mvar mvarId _) => do
    match (← getExprMVarAssignment? mvarId) with
    | some val => if val.isMVar then reduce structNames val else pure val
    | none     => return e
  | e => return e

partial def tryToSynthesizeDefault (structs : Array Struct) (allStructNames : Array Name) (maxDistance : Nat) (fieldName : Name) (mvarId
  let rec loop (i : Nat) (dist : Nat) := do
    if dist > maxDistance then
      pure false
    else if h : i < structs.size then do
      let struct := structs.get ⟨i, h⟩
      let defaultName := struct.structName ++ fieldName ++ `_default
      let env ← getEnv
      match env.find? defaultName with
      | some cinfo@(ConstantInfo.defnInfo defVal) => do
        let mctx ← getMCtx
        let val? ← mkDefaultValue? struct cinfo
        match val? with
        | none     => do setMCtx mctx; loop (i+1) (dist+1)
        | some val => do
          let val ← reduce allStructNames val
          match val.find? fun e => (defaultMissing? e).isSome with
          | some _ => setMCtx mctx; loop (i+1) (dist+1)
          | none   =>
            let mvarDecl ← getMVarDecl mvarId
            let val ← ensureHasType mvarDecl.type val
            assignExprMVar mvarId val
            pure true
      | _ => loop (i+1) dist
    else
      pure false
  loop 0 0

partial def step (struct : Struct) : M Unit :=
  unless (← isRoundDone) do
    withReader (fun ctx => { ctx with structs := ctx.structs.push struct }) do
      for field in struct.fields do
        match field.val with
        | FieldVal.nested struct => step struct
        | _ => match field.expr? with
```

```
        | none      => unreachable!
        | some expr => match defaultMissing? expr with
          | some (Expr.mvar mvarId _) =>
            unless (← isExprMVarAssigned mvarId) do
              let ctx ← read
              if (← withRef field.ref <| tryToSynthesizeDefault ctx.structs ctx.allStructNames ctx.maxDistance (getFieldName field) mv
                modify fun s => { s with progress := true }
          | _ => pure ()

partial def propagateLoop (hierarchyDepth : Nat) (d : Nat) (struct : Struct) : M Unit := do
  match findDefaultMissing? (← getMCtx) struct with
  | none       => pure () -- Done
  | some field =>
    if d > hierarchyDepth then
      throwErrorAt field.ref "field '{getFieldName field}' is missing"
    else withReader (fun ctx => { ctx with maxDistance := d }) do
      modify fun s => { s with progress := false }
      step struct
      if (← get).progress then do
        propagateLoop hierarchyDepth 0 struct
      else
        propagateLoop hierarchyDepth (d+1) struct

def propagate (struct : Struct) : TermElabM Unit :=
  let hierarchyDepth := getHierarchyDepth struct
  let structNames := collectStructNames struct #[]
  (propagateLoop hierarchyDepth 0 struct { allStructNames := structNames }).run' {}

end DefaultFields

private def elabStructInstAux (stx : Syntax) (expectedType? : Option Expr) (source : Source) : TermElabM Expr := do
  let (structName, structType) ← getStructName stx expectedType? source
  unless isStructureLike (← getEnv) structName do
    throwError "invalid \{...} notation, structure type expected{indentExpr structType}"
  match mkStructView stx structName source with
  | Except.error ex  => throwError ex
  | Except.ok struct =>
    let struct ← expandStruct struct
    trace[Elab.struct] "{struct}"
    let (r, struct) ← elabStruct struct expectedType?
    DefaultFields.propagate struct
    pure r

@[builtinTermElab structInst] def elabStructInst : TermElab := fun stx expectedType? => do
  match (← expandNonAtomicExplicitSource stx) with
  | some stxNew => withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
  | none =>
```

```
    let sourceView ← getStructSource stx
    match (← isModifyOp? stx), sourceView with
    | some modifyOp, Source.explicit source _ => elabModifyOp stx modifyOp source expectedType?
    | some _,        _                        => throwError "invalid \{...} notation, explicit source is required when using '[<index>]
    | _,             _                        => elabStructInstAux stx expectedType? sourceView

builtin_initialize registerTraceClass `Elab.struct

end Lean.Elab.Term.StructInst
::::::::::::::
Elab/Structure.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Command
import Lean.Meta.Closure
import Lean.Meta.SizeOf
import Lean.Elab.Command
import Lean.Elab.DeclModifiers
import Lean.Elab.DeclUtil
import Lean.Elab.Inductive
import Lean.Elab.DeclarationRange

namespace Lean.Elab.Command

open Meta

/- Recall that the `structure command syntax is
```
leading_parser (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType >> optional (" :=
```
-/

structure StructCtorView where
  ref       : Syntax
  modifiers : Modifiers
  inferMod  : Bool  -- true if `{}` is used in the constructor declaration
  name      : Name
  declName  : Name

structure StructFieldView where
  ref        : Syntax
  modifiers  : Modifiers
  binderInfo : BinderInfo
```

```
  inferMod   : Bool
  declName   : Name
  name       : Name
  binders    : Syntax
  type?      : Option Syntax
  value?     : Option Syntax

structure StructView where
  ref                : Syntax
  modifiers          : Modifiers
  scopeLevelNames    : List Name  -- All `universe` declarations in the current scope
  allUserLevelNames : List Name  -- `scopeLevelNames` ++ explicit universe parameters provided in the `structure` command
  isClass            : Bool
  declName           : Name
  scopeVars          : Array Expr -- All `variable` declaration in the current scope
  params             : Array Expr -- Explicit parameters provided in the `structure` command
  parents            : Array Syntax
  type               : Syntax
  ctor               : StructCtorView
  fields             : Array StructFieldView

inductive StructFieldKind where
  | newField | fromParent | subobject
  deriving Inhabited

structure StructFieldInfo where
  name     : Name
  declName : Name -- Remark: this field value doesn't matter for fromParent fields.
  fvar     : Expr
  kind     : StructFieldKind
  inferMod : Bool := false
  value?   : Option Expr := none
  deriving Inhabited

def StructFieldInfo.isFromParent (info : StructFieldInfo) : Bool :=
  match info.kind with
  | StructFieldKind.fromParent => true
  | _                          => false

def StructFieldInfo.isSubobject (info : StructFieldInfo) : Bool :=
  match info.kind with
  | StructFieldKind.subobject => true
  | _                         => false

/- Auxiliary declaration for `mkProjections` -/
structure ProjectionInfo where
  declName : Name
```

```
  inferMod : Bool

structure ElabStructResult where
  decl            : Declaration
  projInfos       : List ProjectionInfo
  projInstances   : List Name -- projections (to parent classes) that must be marked as instances.
  mctx            : MetavarContext
  lctx            : LocalContext
  localInsts      : LocalInstances
  defaultAuxDecls : Array (Name × Expr × Expr)

private def defaultCtorName := `mk

/-
The structure constructor syntax is
```
leading_parser try (declModifiers >> ident >> optional inferMod >> " :: ")
```
-/
private def expandCtor (structStx : Syntax) (structModifiers : Modifiers) (structDeclName : Name) : TermElabM StructCtorView := do
  let useDefault := do
    let declName := structDeclName ++ defaultCtorName
    addAuxDeclarationRanges declName structStx[2] structStx[2]
    pure { ref := structStx, modifiers := {}, inferMod := false, name := defaultCtorName, declName := declName }
  if structStx[5].isNone then
    useDefault
  else
    let optCtor := structStx[5][1]
    if optCtor.isNone then
      useDefault
    else
      let ctor := optCtor[0]
      withRef ctor do
      let ctorModifiers ← elabModifiers ctor[0]
      checkValidCtorModifier ctorModifiers
      if ctorModifiers.isPrivate && structModifiers.isPrivate then
        throwError "invalid 'private' constructor in a 'private' structure"
      if ctorModifiers.isProtected && structModifiers.isPrivate then
        throwError "invalid 'protected' constructor in a 'private' structure"
      let inferMod := !ctor[2].isNone
      let name := ctor[1].getId
      let declName := structDeclName ++ name
      let declName ← applyVisibility ctorModifiers.visibility declName
      addDocString' declName ctorModifiers.docString?
      addAuxDeclarationRanges declName ctor[1] ctor[1]
      pure { ref := ctor, name := name, modifiers := ctorModifiers, inferMod := inferMod, declName := declName }

```

```
def checkValidFieldModifier (modifiers : Modifiers) : TermElabM Unit := do
  if modifiers.isNoncomputable then
    throwError "invalid use of 'noncomputable' in field declaration"
  if modifiers.isPartial then
    throwError "invalid use of 'partial' in field declaration"
  if modifiers.isUnsafe then
    throwError "invalid use of 'unsafe' in field declaration"
  if modifiers.attrs.size != 0 then
    throwError "invalid use of attributes in field declaration"
  if modifiers.isPrivate then
    throwError "private fields are not supported yet"

/-
```
def structExplicitBinder := leading_parser atomic (declModifiers true >> "(") >> many1 ident >> optional inferMod >> optDeclSig >> opti
def structImplicitBinder := leading_parser atomic (declModifiers true >> "{") >> many1 ident >> optional inferMod >> declSig >> "}"
def structInstBinder   := leading_parser atomic (declModifiers true >> "[") >> many1 ident >> optional inferMod >> declSig >> "]"
def structSimpleBinder   := leading_parser atomic (declModifiers true >> ident) >> optional inferMod >> optDeclSig >> optional Term.bin
def structFields   := leading_parser many (structExplicitBinder <|> structImplicitBinder <|> structInstBinder)
```
-/
private def expandFields (structStx : Syntax) (structModifiers : Modifiers) (structDeclName : Name) : TermElabM (Array StructFieldView)
  let fieldBinders := if structStx[5].isNone then #[] else structStx[5][2][0].getArgs
  fieldBinders.foldlM (init := #[]) fun (views : Array StructFieldView) fieldBinder => withRef fieldBinder do
    let mut fieldBinder := fieldBinder
    if fieldBinder.getKind == ``Parser.Command.structSimpleBinder then
      fieldBinder := Syntax.node ``Parser.Command.structExplicitBinder
        #[ fieldBinder[0], mkAtomFrom fieldBinder "(", mkNullNode #[ fieldBinder[1] ], fieldBinder[2], fieldBinder[3], fieldBinder[4],
    let k := fieldBinder.getKind
    let binfo ←
      if k == ``Parser.Command.structExplicitBinder then pure BinderInfo.default
      else if k == ``Parser.Command.structImplicitBinder then pure BinderInfo.implicit
      else if k == ``Parser.Command.structInstBinder then pure BinderInfo.instImplicit
      else throwError "unexpected kind of structure field"
    let fieldModifiers ← elabModifiers fieldBinder[0]
    checkValidFieldModifier fieldModifiers
    if fieldModifiers.isPrivate && structModifiers.isPrivate then
      throwError "invalid 'private' field in a 'private' structure"
    if fieldModifiers.isProtected && structModifiers.isPrivate then
      throwError "invalid 'protected' field in a 'private' structure"
    let inferMod        := !fieldBinder[3].isNone
    let (binders, type?) :=
      if binfo == BinderInfo.default then
        expandOptDeclSig fieldBinder[4]
      else
        let (binders, type) := expandDeclSig fieldBinder[4]
        (binders, some type)
```

```
    let value? :=
      if binfo != BinderInfo.default then none
      else
        let optBinderDefault := fieldBinder[5]
        if optBinderDefault.isNone then none
        else
          -- binderDefault := leading_parser " := " >> termParser
          some optBinderDefault[0][1]
    let idents := fieldBinder[2].getArgs
    idents.foldlM (init := views) fun (views : Array StructFieldView) ident => withRef ident do
      let name     := ident.getId
      if isInternalSubobjectFieldName name then
        throwError "invalid field name '{name}', identifiers starting with '_' are reserved to the system"
      let declName := structDeclName ++ name
      let declName ← applyVisibility fieldModifiers.visibility declName
      addDocString' declName fieldModifiers.docString?
      return views.push {
        ref        := ident,
        modifiers  := fieldModifiers,
        binderInfo := binfo,
        inferMod   := inferMod,
        declName   := declName,
        name       := name,
        binders    := binders,
        type?      := type?,
        value?     := value?
      }

private def validStructType (type : Expr) : Bool :=
  match type with
  | Expr.sort .. => true
  | _            => false

private def checkParentIsStructure (parent : Expr) : TermElabM Name :=
  match parent.getAppFn with
  | Expr.const c _ _ => do
    unless isStructure (← getEnv) c do
      throwError "'{c}' is not a structure"
    pure c
  | _ => throwError "expected structure"

private def findFieldInfo? (infos : Array StructFieldInfo) (fieldName : Name) : Option StructFieldInfo :=
  infos.find? fun info => info.name == fieldName

private def containsFieldName (infos : Array StructFieldInfo) (fieldName : Name) : Bool :=
  (findFieldInfo? infos fieldName).isSome
```

```
private partial def processSubfields (structDeclName : Name) (parentFVar : Expr) (parentStructName : Name) (subfieldNames : Array Name)
    (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α) : TermElabM α :=
  let rec loop (i : Nat) (infos : Array StructFieldInfo) := do
    if h : i < subfieldNames.size then
      let subfieldName := subfieldNames.get ⟨i, h⟩
      if containsFieldName infos subfieldName then
        throwError "field '{subfieldName}' from '{parentStructName}' has already been declared"
      let val  ← mkProjection parentFVar subfieldName
      let type ← inferType val
      withLetDecl subfieldName type val fun subfieldFVar =>
        /- The following `declName` is only used for creating the `_default` auxiliary declaration name when
           its default value is overwritten in the structure. -/
        let declName := structDeclName ++ subfieldName
        let infos := infos.push { name := subfieldName, declName := declName, fvar := subfieldFVar, kind := StructFieldKind.fromParent
        loop (i+1) infos
    else
      k infos
  loop 0 infos

private partial def withParents (view : StructView) (i : Nat) (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α)
  if h : i < view.parents.size then
    let parentStx := view.parents.get ⟨i, h⟩
    withRef parentStx do
    let parent ← Term.elabType parentStx
    let parentName ← checkParentIsStructure parent
    let toParentName := Name.mkSimple $ "to" ++ parentName.eraseMacroScopes.getString! -- erase macro scopes?
    if containsFieldName infos toParentName then
      throwErrorAt parentStx "field '{toParentName}' has already been declared"
    let env ← getEnv
    let binfo := if view.isClass && isClass env parentName then BinderInfo.instImplicit else BinderInfo.default
    withLocalDecl toParentName binfo parent fun parentFVar =>
      let infos := infos.push { name := toParentName, declName := view.declName ++ toParentName, fvar := parentFVar, kind := StructFiel
      let subfieldNames := getStructureFieldsFlattened env parentName
      processSubfields view.declName parentFVar parentName subfieldNames infos fun infos => withParents view (i+1) infos k
  else
    k infos

private def elabFieldTypeValue (view : StructFieldView) : TermElabM (Option Expr × Option Expr) := do
  Term.withAutoBoundImplicit <| Term.elabBinders view.binders.getArgs fun params => do
    match view.type? with
    | none         =>
      match view.value? with
      | none        => return (none, none)
      | some valStx =>
        Term.synthesizeSyntheticMVarsNoPostponing
        let params ← Term.addAutoBoundImplicits params
        let value ← Term.elabTerm valStx none
```

```
        let value ← mkLambdaFVars params value
        return (none, value)
    | some typeStx =>
      let type ← Term.elabType typeStx
      Term.synthesizeSyntheticMVarsNoPostponing
      let params ← Term.addAutoBoundImplicits params
      match view.value? with
      | none        =>
        let type  ← mkForallFVars params type
        return (type, none)
      | some valStx =>
        let value ← Term.elabTermEnsuringType valStx type
        Term.synthesizeSyntheticMVarsNoPostponing
        let type  ← mkForallFVars params type
        let value ← mkLambdaFVars params value
        return (type, value)

private partial def withFields
    (views : Array StructFieldView) (i : Nat) (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α) : TermElabM α :=
  if h : i < views.size then
    let view := views.get ⟨i, h⟩
    withRef view.ref $
    match findFieldInfo? infos view.name with
    | none      => do
      let (type?, value?) ← elabFieldTypeValue view
      match type?, value? with
      | none,      none => throwError "invalid field, type expected"
      | some type, _    =>
        withLocalDecl view.name view.binderInfo type fun fieldFVar =>
          let infos := infos.push { name := view.name, declName := view.declName, fvar := fieldFVar, value? := value?,
                                    kind := StructFieldKind.newField, inferMod := view.inferMod }
          withFields views (i+1) infos k
      | none, some value =>
        let type ← inferType value
        withLocalDecl view.name view.binderInfo type fun fieldFVar =>
          let infos := infos.push { name := view.name, declName := view.declName, fvar := fieldFVar, value? := value,
                                    kind := StructFieldKind.newField, inferMod := view.inferMod }
          withFields views (i+1) infos k
    | some info =>
      match info.kind with
      | StructFieldKind.newField   => throwError "field '{view.name}' has already been declared"
      | StructFieldKind.fromParent =>
        match view.value? with
        | none       => throwError "field '{view.name}' has been declared in parent structure"
        | some valStx => do
          if let some type := view.type? then
            throwErrorAt type "omit field '{view.name}' type to set default value"
```

```
        else
          let mut valStx := valStx
          if view.binders.getArgs.size > 0 then
            valStx ← `(fun $(view.binders.getArgs)* => $valStx:term)
          let fvarType ← inferType info.fvar
          let value ← Term.elabTermEnsuringType valStx fvarType
          let infos := infos.push { info with value? := value }
          withFields views (i+1) infos k
    | StructFieldKind.subobject => unreachable!
  else
    k infos

private def getResultUniverse (type : Expr) : TermElabM Level := do
  let type ← whnf type
  match type with
  | Expr.sort u _ => pure u
  | _             => throwError "unexpected structure resulting type"

private def collectUsed (params : Array Expr) (fieldInfos : Array StructFieldInfo) : StateRefT CollectFVars.State MetaM Unit := do
  params.forM fun p => do
    let type ← inferType p
    Term.collectUsedFVars type
  fieldInfos.forM fun info => do
    let fvarType ← inferType info.fvar
    Term.collectUsedFVars fvarType
    match info.value? with
    | none       => pure ()
    | some value => Term.collectUsedFVars value

private def removeUnused (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo)
    : TermElabM (LocalContext × LocalInstances × Array Expr) := do
  let (_, used) ← (collectUsed params fieldInfos).run {}
  Term.removeUnused scopeVars used

private def withUsed {α} (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo) (k : Array Expr → TermElabM
    : TermElabM α := do
  let (lctx, localInsts, vars) ← removeUnused scopeVars params fieldInfos
  withLCtx lctx localInsts $ k vars

private def levelMVarToParamFVar (fvar : Expr) : StateRefT Nat TermElabM Unit := do
  let type ← inferType fvar
  discard <| Term.levelMVarToParam' type

private def levelMVarToParamFVars (fvars : Array Expr) : StateRefT Nat TermElabM Unit :=
  fvars.forM levelMVarToParamFVar

private def levelMVarToParamAux (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo)
```

```
    : StateRefT Nat TermElabM (Array StructFieldInfo) := do
  levelMVarToParamFVars scopeVars
  levelMVarToParamFVars params
  fieldInfos.mapM fun info => do
    levelMVarToParamFVar info.fvar
    match info.value? with
    | none       => pure info
    | some value =>
      let value ← Term.levelMVarToParam' value
      pure { info with value? := value }

private def levelMVarToParam (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo) : TermElabM (Array Str
  (levelMVarToParamAux scopeVars params fieldInfos).run' 1

private partial def collectUniversesFromFields (r : Level) (rOffset : Nat) (fieldInfos : Array StructFieldInfo) : TermElabM (Array Leve
  fieldInfos.foldlM (init := #[]) fun (us : Array Level) (info : StructFieldInfo) => do
    let type ← inferType info.fvar
    let u ← getLevel type
    let u ← instantiateLevelMVars u
    accLevelAtCtor u r rOffset us

private def updateResultingUniverse (fieldInfos : Array StructFieldInfo) (type : Expr) : TermElabM Expr := do
  let r ← getResultUniverse type
  let rOffset : Nat   := r.getOffset
  let r       : Level := r.getLevelOffset
  match r with
  | Level.mvar mvarId _ =>
    let us ← collectUniversesFromFields r rOffset fieldInfos
    let rNew := mkResultUniverse us rOffset
    assignLevelMVar mvarId rNew
    instantiateMVars type
  | _ => throwError "failed to compute resulting universe level of structure, provide universe explicitly"

private def collectLevelParamsInFVar (s : CollectLevelParams.State) (fvar : Expr) : TermElabM CollectLevelParams.State := do
  let type ← inferType fvar
  let type ← instantiateMVars type
  pure $ collectLevelParams s type

private def collectLevelParamsInFVars (fvars : Array Expr) (s : CollectLevelParams.State) : TermElabM CollectLevelParams.State :=
  fvars.foldlM collectLevelParamsInFVar s

private def collectLevelParamsInStructure (structType : Expr) (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array Struct
    : TermElabM (Array Name) := do
  let s := collectLevelParams {} structType
  let s ← collectLevelParamsInFVars scopeVars s
  let s ← collectLevelParamsInFVars params s
  let s ← fieldInfos.foldlM (fun (s : CollectLevelParams.State) info => collectLevelParamsInFVar s info.fvar) s
```

```
  pure s.params

private def addCtorFields (fieldInfos : Array StructFieldInfo) : Nat → Expr → TermElabM Expr
  | 0,   type => pure type
  | i+1, type => do
    let info := fieldInfos[i]
    let decl ← Term.getFVarLocalDecl! info.fvar
    let type ← instantiateMVars type
    let type := type.abstract #[info.fvar]
    match info.kind with
    | StructFieldKind.fromParent =>
      let val := decl.value
      addCtorFields fieldInfos i (type.instantiate1 val)
    | StructFieldKind.subobject =>
      let n := mkInternalSubobjectFieldName $ decl.userName
      addCtorFields fieldInfos i (mkForall n decl.binderInfo decl.type type)
    | StructFieldKind.newField =>
      addCtorFields fieldInfos i (mkForall decl.userName decl.binderInfo decl.type type)

private def mkCtor (view : StructView) (levelParams : List Name) (params : Array Expr) (fieldInfos : Array StructFieldInfo) : TermElabM
  withRef view.ref do
  let type := mkAppN (mkConst view.declName (levelParams.map mkLevelParam)) params
  let type ← addCtorFields fieldInfos fieldInfos.size type
  let type ← mkForallFVars params type
  let type ← instantiateMVars type
  let type := type.inferImplicit params.size !view.ctor.inferMod
  pure { name := view.ctor.declName, type := type }

@[extern "lean_mk_projections"]
private constant mkProjections (env : Environment) (structName : Name) (projs : List ProjectionInfo) (isClass : Bool) : Except KernelEx

private def addProjections (structName : Name) (projs : List ProjectionInfo) (isClass : Bool) : TermElabM Unit := do
  let env ← getEnv
  match mkProjections env structName projs isClass with
  | Except.ok env   => setEnv env
  | Except.error ex => throwKernelException ex

private def mkAuxConstructions (declName : Name) : TermElabM Unit := do
  let env ← getEnv
  let hasUnit := env.contains `PUnit
  let hasEq   := env.contains `Eq
  let hasHEq  := env.contains `HEq
  mkRecOn declName
  if hasUnit then mkCasesOn declName
  if hasUnit && hasEq && hasHEq then mkNoConfusion declName

private def addDefaults (lctx : LocalContext) (defaultAuxDecls : Array (Name × Expr × Expr)) : TermElabM Unit := do
```

```
  let localInsts ← getLocalInstances
  withLCtx lctx localInsts do
    defaultAuxDecls.forM fun (declName, type, value) => do
      let value ← instantiateMVars value
      if value.hasExprMVar then
        throwError "invalid default value for field, it contains metavariables{indentExpr value}"
      /- The identity function is used as "marker". -/
      let value ← mkId value
      discard <| mkAuxDefinition declName type value (zeta := true)
      setReducibleAttribute declName

private def elabStructureView (view : StructView) : TermElabM Unit := do
  view.fields.forM fun field => do
    if field.declName == view.ctor.declName then
      throwErrorAt field.ref "invalid field name '{field.name}', it is equal to structure constructor name"
    addAuxDeclarationRanges field.declName field.ref field.ref
  let numExplicitParams := view.params.size
  let type ← Term.elabType view.type
  unless validStructType type do throwErrorAt view.type "expected Type"
  withRef view.ref do
  withParents view 0 #[] fun fieldInfos =>
  withFields view.fields 0 fieldInfos fun fieldInfos => do
    Term.synthesizeSyntheticMVarsNoPostponing
    let u ← getResultUniverse type
    let inferLevel ← shouldInferResultUniverse u
    withUsed view.scopeVars view.params fieldInfos $ fun scopeVars => do
      let numParams := scopeVars.size + numExplicitParams
      let fieldInfos ← levelMVarToParam scopeVars view.params fieldInfos
      let type ← withRef view.ref do
        if inferLevel then
          updateResultingUniverse fieldInfos type
        else
          checkResultingUniverse (← getResultUniverse type)
          pure type
      trace[Elab.structure] "type: {type}"
      let usedLevelNames ← collectLevelParamsInStructure type scopeVars view.params fieldInfos
      match sortDeclLevelParams view.scopeLevelNames view.allUserLevelNames usedLevelNames with
      | Except.error msg      => withRef view.ref <| throwError msg
      | Except.ok levelParams =>
        let params := scopeVars ++ view.params
        let ctor ← mkCtor view levelParams params fieldInfos
        let type ← mkForallFVars params type
        let type ← instantiateMVars type
        let indType := { name := view.declName, type := type, ctors := [ctor] : InductiveType }
        let decl    := Declaration.inductDecl levelParams params.size [indType] view.modifiers.isUnsafe
        Term.ensureNoUnassignedMVars decl
        addDecl decl
```

```
        let projInfos := (fieldInfos.filter fun (info : StructFieldInfo) => !info.isFromParent).toList.map fun (info : StructFieldInfo)
          { declName := info.declName, inferMod := info.inferMod : ProjectionInfo }
        addProjections view.declName projInfos view.isClass
        mkAuxConstructions view.declName
        let instParents ← fieldInfos.filterM fun info => do
          let decl ← Term.getFVarLocalDecl! info.fvar
          pure (info.isSubobject && decl.binderInfo.isInstImplicit)
        let projInstances := instParents.toList.map fun info => info.declName
        Term.applyAttributesAt view.declName view.modifiers.attrs AttributeApplicationTime.afterTypeChecking
        projInstances.forM fun declName => addInstance declName AttributeKind.global (eval_prio default)
        let lctx ← getLCtx
        let fieldsWithDefault := fieldInfos.filter fun info => info.value?.isSome
        let defaultAuxDecls ← fieldsWithDefault.mapM fun info => do
          let type ← inferType info.fvar
          pure (info.declName ++ `_default, type, info.value?.get!)
        /- The `lctx` and `defaultAuxDecls` are used to create the auxiliary `_default` declarations
           The parameters `params` for these definitions must be marked as implicit, and all others as explicit. -/
        let lctx :=
          params.foldl (init := lctx) fun (lctx : LocalContext) (p : Expr) =>
            lctx.setBinderInfo p.fvarId! BinderInfo.implicit
        let lctx :=
          fieldInfos.foldl (init := lctx) fun (lctx : LocalContext) (info : StructFieldInfo) =>
            if info.isFromParent then lctx -- `fromParent` fields are elaborated as let-decls, and are zeta-expanded when creating `_de
            else lctx.setBinderInfo info.fvar.fvarId! BinderInfo.default
        addDefaults lctx defaultAuxDecls

/-
leading_parser (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType >> " := " >> opti

where
def «extends» := leading_parser " extends " >> sepBy1 termParser ", "
def typeSpec := leading_parser " : " >> termParser
def optType : Parser := optional typeSpec

def structFields         := leading_parser many (structExplicitBinder <|> structImplicitBinder <|> structInstBinder)
def structCtor           := leading_parser try (declModifiers >> ident >> optional inferMod >> " :: ")

-/
def elabStructure (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
  checkValidInductiveModifier modifiers
  let isClass   := stx[0].getKind == ``Parser.Command.classTk
  let modifiers := if isClass then modifiers.addAttribute { name := `class } else modifiers
  let declId    := stx[1]
  let params    := stx[2].getArgs
  let exts      := stx[3]
  let parents   := if exts.isNone then #[] else exts[0][1].getSepArgs
  let optType   := stx[4]
```

```
  let derivingClassViews ← getOptDerivingClasses stx[6]
  let type ← if optType.isNone then `(Sort _) else pure optType[0][1]
  let declName ←
    runTermElabM none fun scopeVars => do
      let scopeLevelNames ← Term.getLevelNames
      let ⟨name, declName, allUserLevelNames⟩ ← Elab.expandDeclId (← getCurrNamespace) scopeLevelNames declId modifiers
      addDeclarationRanges declName stx
      Term.withDeclName declName do
        let ctor ← expandCtor stx modifiers declName
        let fields ← expandFields stx modifiers declName
        Term.withLevelNames allUserLevelNames <| Term.withAutoBoundImplicit <|
          Term.elabBinders params fun params => do
            Term.synthesizeSyntheticMVarsNoPostponing
            let params ← Term.addAutoBoundImplicits params
            let allUserLevelNames ← Term.getLevelNames
            elabStructureView {
              ref               := stx
              modifiers         := modifiers
              scopeLevelNames   := scopeLevelNames
              allUserLevelNames := allUserLevelNames
              declName          := declName
              isClass           := isClass
              scopeVars         := scopeVars
              params            := params
              parents           := parents
              type              := type
              ctor              := ctor
              fields            := fields
            }
            unless isClass do
              mkSizeOfInstances declName
            return declName
  derivingClassViews.forM fun view => view.applyHandlers #[declName]

builtin_initialize registerTraceClass `Elab.structure

end Lean.Elab.Command
::::::::::::::
Elab/Syntax.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Elab.Command
import Lean.Parser.Syntax
```

```
namespace Lean.Elab.Term
/-
Expand `optional «precedence»` where
 «precedence» := leading_parser " : " >> precedenceParser -/
def expandOptPrecedence (stx : Syntax) : MacroM (Option Nat) :=
  if stx.isNone then
    return none
  else
    return some (← evalPrec stx[0][1])

private def mkParserSeq (ds : Array Syntax) : TermElabM Syntax := do
  if ds.size == 0 then
    throwUnsupportedSyntax
  else if ds.size == 1 then
    pure ds[0]
  else
    let mut r := ds[0]
    for d in ds[1:ds.size] do
      r ← `(ParserDescr.binary `andthen $r $d)
    return r

structure ToParserDescrContext where
  catName  : Name
  first    : Bool
  leftRec  : Bool -- true iff left recursion is allowed
  /- See comment at `Parser.ParserCategory`. -/
  behavior : Parser.LeadingIdentBehavior

abbrev ToParserDescrM := ReaderT ToParserDescrContext (StateRefT (Option Nat) TermElabM)
private def markAsTrailingParser (lhsPrec : Nat) : ToParserDescrM Unit := set (some lhsPrec)

@[inline] private def withNotFirst {α} (x : ToParserDescrM α) : ToParserDescrM α :=
  withReader (fun ctx => { ctx with first := false }) x

@[inline] private def withNestedParser {α} (x : ToParserDescrM α) : ToParserDescrM α :=
  withReader (fun ctx => { ctx with leftRec := false, first := false }) x

def checkLeftRec (stx : Syntax) : ToParserDescrM Bool := do
  let ctx ← read
  unless ctx.first && stx.getKind == `Lean.Parser.Syntax.cat do
    return false
  let cat := stx[0].getId.eraseMacroScopes
  unless cat == ctx.catName do
    return false
  let prec? ← liftMacroM <| expandOptPrecedence stx[1]
  unless ctx.leftRec do
```

```
    throwErrorAt stx[3] "invalid occurrence of '{cat}', parser algorithm does not allow this form of left recursion"
  markAsTrailingParser (prec?.getD 0)
  return true

/--
  Given a `stx` of category `syntax`, return a pair `(newStx, lhsPrec?)`,
  where `newStx` is of category `term`. After elaboration, `newStx` should have type
  `TrailingParserDescr` if `lhsPrec?.isSome`, and `ParserDescr` otherwise. -/
partial def toParserDescr (stx : Syntax) (catName : Name) : TermElabM (Syntax × Option Nat) := do
  let env ← getEnv
  let behavior := Parser.leadingIdentBehavior env catName
  (process stx { catName := catName, first := true, leftRec := true, behavior := behavior }).run none
where
  process (stx : Syntax) : ToParserDescrM Syntax := withRef stx do
    let kind := stx.getKind
    if kind == nullKind then
      processSeq stx
    else if kind == choiceKind then
      process stx[0]
    else if kind == `Lean.Parser.Syntax.paren then
      process stx[1]
    else if kind == `Lean.Parser.Syntax.cat then
      processNullaryOrCat stx
    else if kind == `Lean.Parser.Syntax.unary then
      processUnary stx
    else if kind == `Lean.Parser.Syntax.binary then
      processBinary stx
    else if kind == `Lean.Parser.Syntax.sepBy then
      processSepBy stx
    else if kind == `Lean.Parser.Syntax.sepBy1 then
      processSepBy1 stx
    else if kind == `Lean.Parser.Syntax.atom then
      processAtom stx
    else if kind == `Lean.Parser.Syntax.nonReserved then
      processNonReserved stx
    else
      let stxNew? ← liftM (liftMacroM (expandMacro? stx) : TermElabM _)
      match stxNew? with
      | some stxNew => process stxNew
      | none => throwErrorAt stx "unexpected syntax kind of category `syntax`: {kind}"

  /- Sequence (aka NullNode) -/
  processSeq (stx : Syntax) := do
    let args := stx.getArgs
    if (← checkLeftRec stx[0]) then
      if args.size == 1 then throwErrorAt stx "invalid atomic left recursive syntax"
      let args := args.eraseIdx 0
```

```
    let args ← args.mapM fun arg => withNestedParser do process arg
    mkParserSeq args
  else
    let args ← args.mapIdxM fun i arg => withReader (fun ctx => { ctx with first := ctx.first && i.val == 0 }) do process arg
    mkParserSeq args

/- Resolve the given parser name and return a list of candidates.
   Each candidate is a pair `(resolvedParserName, isDescr)`.
   `isDescr == true` if the type of `resolvedParserName` is a `ParserDescr`. -/
resolveParserName (parserName : Name) : ToParserDescrM (List (Name × Bool)) := do
  try
    let candidates ← resolveGlobalConstWithInfos (← getRef) parserName
    /- Convert `candidates` in a list of pairs `(c, isDescr)`, where `c` is the parser name,
       and `isDescr` is true iff `c` has type `Lean.ParserDescr` or `Lean.TrailingParser` -/
    let env ← getEnv
    candidates.filterMap fun c =>
       match env.find? c with
       | none      => none
       | some info =>
         match info.type with
        | Expr.const `Lean.Parser.TrailingParser _ _ => (c, false)
        | Expr.const `Lean.Parser.Parser _ _         => (c, false)
        | Expr.const `Lean.ParserDescr _ _           => (c, true)
        | Expr.const `Lean.TrailingParserDescr _ _   => (c, true)
        | _                                          => none
  catch _ => return []

ensureNoPrec (stx : Syntax) :=
  unless stx[1].isNone do
    throwErrorAt stx[1] "unexpected precedence"

processParserCategory (stx : Syntax) := do
  let catName := stx[0].getId.eraseMacroScopes
  if (← read).first && catName == (← read).catName then
    throwErrorAt stx "invalid atomic left recursive syntax"
  let prec? ← liftMacroM <| expandOptPrecedence stx[1]
  let prec := prec?.getD 0
  `(ParserDescr.cat $(quote catName) $(quote prec))

processNullaryOrCat (stx : Syntax) := do
  let id := stx[0].getId.eraseMacroScopes
  match (← withRef stx[0] <| resolveParserName id) with
  | [(c, true)]      => ensureNoPrec stx; return mkIdentFrom stx c
  | [(c, false)]     => ensureNoPrec stx; `(ParserDescr.parser $(quote c))
  | cs@(_ :: _ :: _) => throwError "ambiguous parser declaration {cs.map (·.1)}"
  | [] =>
    if Parser.isParserCategory (← getEnv) id then
```

```
      processParserCategory stx
    else if (← Parser.isParserAlias id) then
      ensureNoPrec stx
      Parser.ensureConstantParserAlias id
      `(ParserDescr.const $(quote id))
    else
      throwError "unknown parser declaration/category/alias '{id}'"

  processUnary (stx : Syntax) := do
    let aliasName := (stx[0].getId).eraseMacroScopes
    Parser.ensureUnaryParserAlias aliasName
    let d ← withNestedParser do process stx[2]
    `(ParserDescr.unary $(quote aliasName) $d)

  processBinary (stx : Syntax) := do
    let aliasName := (stx[0].getId).eraseMacroScopes
    Parser.ensureBinaryParserAlias aliasName
    let d₁ ← withNestedParser do process stx[2]
    let d₂ ← withNestedParser do process stx[4]
    `(ParserDescr.binary $(quote aliasName) $d₁ $d₂)

  processSepBy (stx : Syntax) := do
    let p ← withNestedParser $ process stx[1]
    let sep := stx[3]
    let psep ← if stx[4].isNone then `(ParserDescr.symbol $sep) else process stx[4][1]
    let allowTrailingSep := !stx[5].isNone
    `(ParserDescr.sepBy $p $sep $psep $(quote allowTrailingSep))

  processSepBy1 (stx : Syntax) := do
    let p ← withNestedParser do process stx[1]
    let sep := stx[3]
    let psep ← if stx[4].isNone then `(ParserDescr.symbol $sep) else process stx[4][1]
    let allowTrailingSep := !stx[5].isNone
    `(ParserDescr.sepBy1 $p $sep $psep $(quote allowTrailingSep))

  processAtom (stx : Syntax) := do
    match stx[0].isStrLit? with
    | some atom =>
      /- For syntax categories where initialized with `LeadingIdentBehavior` different from default (e.g., `tactic`), we automatically
         the first symbol as nonReserved. -/
      if (← read).behavior != Parser.LeadingIdentBehavior.default && (← read).first then
        `(ParserDescr.nonReservedSymbol $(quote atom) false)
      else
        `(ParserDescr.symbol $(quote atom))
    | none => throwUnsupportedSyntax

  processNonReserved (stx : Syntax) := do
```

```
    match stx[1].isStrLit? with
    | some atom => `(ParserDescr.nonReservedSymbol $(quote atom) false)
    | none      => throwUnsupportedSyntax


end Term

namespace Command
open Lean.Syntax
open Lean.Parser.Term hiding macroArg
open Lean.Parser.Command

private def getCatSuffix (catName : Name) : String :=
  match catName with
  | Name.str _ s _ => s
  | _              => unreachable!

private def declareSyntaxCatQuotParser (catName : Name) : CommandElabM Unit := do
  let quotSymbol := "`(" ++ getCatSuffix catName ++ "|"
  let name := catName ++ `quot
  -- TODO(Sebastian): this might confuse the pretty printer, but it lets us reuse the elaborator
  let kind := ``Lean.Parser.Term.quot
  let cmd ← `(
    @[termParser] def $(mkIdent name) : Lean.ParserDescr :=
      Lean.ParserDescr.node $(quote kind) $(quote Lean.Parser.maxPrec)
        (Lean.ParserDescr.binary `andthen (Lean.ParserDescr.symbol $(quote quotSymbol))
          (Lean.ParserDescr.binary `andthen (Lean.ParserDescr.cat $(quote catName) 0) (Lean.ParserDescr.symbol ")"))))
  elabCommand cmd

@[builtinCommandElab syntaxCat] def elabDeclareSyntaxCat : CommandElab := fun stx => do
  let catName  := stx[1].getId
  let attrName := catName.appendAfter "Parser"
  let env ← getEnv
  let env ← liftIO $ Parser.registerParserCategory env attrName catName
  setEnv env
  declareSyntaxCatQuotParser catName

/--
  Auxiliary function for creating declaration names from parser descriptions.
  Example:
  Given
  ```
  syntax term "+" term : term
  syntax "[" sepBy(term, ", ") "]" : term
  ```
  It generates the names `term_+_` and `term[_,]`
-/
```

```
partial def mkNameFromParserSyntax (catName : Name) (stx : Syntax) : CommandElabM Name :=
  mkUnusedBaseName <| Name.mkSimple <| appendCatName <| visit stx ""
where
  visit (stx : Syntax) (acc : String) : String :=
    match stx.isStrLit? with
    | some val => acc ++ (val.trim.map fun c => if c.isWhitespace then '_' else c).capitalize
    | none =>
      match stx with
      | Syntax.node k args =>
        if k == `Lean.Parser.Syntax.cat then
          acc ++ "_"
        else
          args.foldl (init := acc) fun acc arg => visit arg acc
      | Syntax.ident ..    => acc
      | Syntax.atom ..     => acc
      | Syntax.missing     => acc

  appendCatName (str : String) :=
    match catName with
    | Name.str _ s _ => s ++ str
    | _ => str

/- We assume a new syntax can be treated as an atom when it starts and ends with a token.
   Here are examples of atom-like syntax.
   ```
   syntax "(" term ")" : term
   syntax "[" (sepBy term ",") "]" : term
   syntax "foo" : term
   ```
 -/
private partial def isAtomLikeSyntax (stx : Syntax) : Bool :=
  let kind := stx.getKind
  if kind == nullKind then
    isAtomLikeSyntax stx[0] && isAtomLikeSyntax stx[stx.getNumArgs - 1]
  else if kind == choiceKind then
    isAtomLikeSyntax stx[0] -- see toParserDescr
  else if kind == `Lean.Parser.Syntax.paren then
    isAtomLikeSyntax stx[1]
  else
    kind == `Lean.Parser.Syntax.atom

@[builtinCommandElab «syntax»] def elabSyntax : CommandElab := fun stx => do
  let `($attrKind:attrKind syntax $[: $prec? ]? $[(name := $name?)]? $[(priority := $prio?)]? $[$ps:stx]* : $catStx) ← pure stx
    | throwUnsupportedSyntax
  let cat := catStx.getId.eraseMacroScopes
  unless (Parser.isParserCategory (← getEnv) cat) do
    throwErrorAt catStx "unknown category '{cat}'"
```

```
  let syntaxParser := mkNullNode ps
  -- If the user did not provide an explicit precedence, we assign `maxPrec` to atom-like syntax and `leadPrec` otherwise.
  let precDefault  := if isAtomLikeSyntax syntaxParser then Parser.maxPrec else Parser.leadPrec
  let prec ← match prec? with
    | some prec => liftMacroM <| evalPrec prec
    | none      => precDefault
  let name ← match name? with
    | some name => pure name.getId
    | none => mkNameFromParserSyntax cat syntaxParser
  let prio ← liftMacroM <| evalOptPrio prio?
  let stxNodeKind := (← getCurrNamespace) ++ name
  let catParserId := mkIdentFrom stx (cat.appendAfter "Parser")
  let (val, lhsPrec?) ← runTermElabM none fun _ => Term.toParserDescr syntaxParser cat
  let declName := mkIdentFrom stx name
  let d ←
    if let some lhsPrec := lhsPrec? then
      `(@[$attrKind:attrKind $catParserId:ident $(quote prio):numLit] def $declName : Lean.TrailingParserDescr :=
        ParserDescr.trailingNode $(quote stxNodeKind) $(quote prec) $(quote lhsPrec) $val)
    else
      `(@[$attrKind:attrKind $catParserId:ident $(quote prio):numLit] def $declName : Lean.ParserDescr :=
        ParserDescr.node $(quote stxNodeKind) $(quote prec) $val)
  trace `Elab fun _ => d
  withMacroExpansion stx d <| elabCommand d

/-
def syntaxAbbrev  := leading_parser "syntax " >> ident >> " := " >> many1 syntaxParser
-/
@[builtinCommandElab «syntaxAbbrev»] def elabSyntaxAbbrev : CommandElab := fun stx => do
  let declName := stx[1]
  -- TODO: nonatomic names
  let (val, _) ← runTermElabM none $ fun _ => Term.toParserDescr stx[3] Name.anonymous
  let stxNodeKind := (← getCurrNamespace) ++ declName.getId
  let stx' ← `(def $declName : Lean.ParserDescr := ParserDescr.nodeWithAntiquot $(quote (toString declName.getId)) $(quote stxNodeKind)
  withMacroExpansion stx stx' $ elabCommand stx'

private def checkRuleKind (given expected : SyntaxNodeKind) : Bool :=
  given == expected || given == expected ++ `antiquot

/-
  Remark: `k` is the user provided kind with the current namespace included.
  Recall that syntax node kinds contain the current namespace.
-/
def elabMacroRulesAux (k : SyntaxNodeKind) (alts : Array Syntax) : CommandElabM Syntax := do
  let alts ← alts.mapM fun alt => match alt with
    | `(matchAltExpr| | $pats,* => $rhs) => do
      let pat := pats.elemsAndSeps[0]
      if !pat.isQuot then
```

```
        throwUnsupportedSyntax
      let quoted := getQuotContent pat
      let k' := quoted.getKind
      if checkRuleKind k' k then
        pure alt
      else if k' == choiceKind then
         match quoted.getArgs.find? fun quotAlt => checkRuleKind quotAlt.getKind k with
         | none        => throwErrorAt alt "invalid macro_rules alternative, expected syntax node kind '{k}'"
         | some quoted =>
           let pat := pat.setArg 1 quoted
           let pats := pats.elemsAndSeps.set! 0 pat
           `(matchAltExpr| | $pats,* => $rhs)
      else
        throwErrorAt alt "invalid macro_rules alternative, unexpected syntax node kind '{k'}'"
    | _ => throwUnsupportedSyntax
  `(@[macro $(Lean.mkIdent k)] def myMacro : Macro :=
     fun $alts:matchAlt* | _ => throw Lean.Macro.Exception.unsupportedSyntax)

def inferMacroRulesAltKind : Syntax → CommandElabM SyntaxNodeKind
  | `(matchAltExpr| | $pats,* => $rhs) => do
    let pat := pats.elemsAndSeps[0]
    if !pat.isQuot then
      throwUnsupportedSyntax
    let quoted := getQuotContent pat
    pure quoted.getKind
  | _ => throwUnsupportedSyntax

def elabNoKindMacroRulesAux (alts : Array Syntax) : CommandElabM Syntax := do
  let mut k ← inferMacroRulesAltKind alts[0]
  if k.isStr && k.getString! == "antiquot" then
    k := k.getPrefix
  if k == choiceKind then
    throwErrorAt alts[0]
      "invalid macro_rules alternative, multiple interpretations for pattern (solution: specify node kind using `macro_rules [<kind>] .
  else
    let altsK    ← alts.filterM fun alt => return checkRuleKind (← inferMacroRulesAltKind alt) k
    let altsNotK ← alts.filterM fun alt => return !checkRuleKind (← inferMacroRulesAltKind alt) k
    let defCmd   ← elabMacroRulesAux k altsK
    if altsNotK.isEmpty then
      pure defCmd
    else
      `($defCmd:command macro_rules $altsNotK:matchAlt*)

@[builtinCommandElab «macro_rules»] def elabMacroRules : CommandElab :=
  adaptExpander fun stx => match stx with
  | `(macro_rules $alts:matchAlt*)                    => elabNoKindMacroRulesAux alts
  | `(macro_rules (kind := $kind) | $x:ident => $rhs) => `(@[macro $kind] def myMacro : Macro := fun $x:ident => $rhs)
```

```
  | `(macro_rules (kind := $kind) $alts:matchAlt*)    => do elabMacroRulesAux ((← getCurrNamespace) ++ kind.getId) alts
  | _                                                 => throwUnsupportedSyntax

@[builtinMacro Lean.Parser.Command.mixfix] def expandMixfix : Macro := fun stx =>
  withAttrKindGlobal stx fun stx => do
    match stx with
    | `(infixl $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
      let prec1 := quote <| (← evalOptPrec prec) + 1
      `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs$[:$prec]? $op:strLit rhs:$prec1 => $f lhs rhs)
    | `(infix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
      let prec1 := quote <| (← evalOptPrec prec) + 1
      `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs:$prec1 $op:strLit rhs:$prec1 => $f lhs rhs)
    | `(infixr $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
      let prec1 := quote <| (← evalOptPrec prec) + 1
      `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs:$prec1 $op:strLit rhs $[: $prec]? => $f lhs rhs)
    | `(prefix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
      `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op:strLit arg $[: $prec]? => $f arg)
    | `(postfix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
      `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? arg$[:$prec]? $op:strLit => $f arg)
    | _ => Macro.throwUnsupported
where
  -- set "global" `attrKind`, apply `f`, and restore `attrKind` to result
  withAttrKindGlobal stx f := do
    let attrKind := stx[0]
    let stx  := stx.setArg 0 mkAttrKindGlobal
    let stx ← f stx
    return stx.setArg 0 attrKind

/- Wrap all occurrences of the given `ident` nodes in antiquotations -/
private partial def antiquote (vars : Array Syntax) : Syntax → Syntax
  | stx => match stx with
  | `($id:ident) =>
    if (vars.findIdx? (fun var => var.getId == id.getId)).isSome then
      mkAntiquotNode id
    else
      stx
  | _ => match stx with
    | Syntax.node k args => Syntax.node k (args.map (antiquote vars))
    | stx => stx

/- Convert `notation` command lhs item into a `syntax` command item -/
def expandNotationItemIntoSyntaxItem (stx : Syntax) : CommandElabM Syntax :=
  let k := stx.getKind
  if k == `Lean.Parser.Command.identPrec then
    pure $ Syntax.node `Lean.Parser.Syntax.cat #[mkIdentFrom stx `term,  stx[1]]
  else if k == strLitKind then
    pure $ Syntax.node `Lean.Parser.Syntax.atom #[stx]
```

```lean
  else
    throwUnsupportedSyntax

def strLitToPattern (stx: Syntax) : MacroM Syntax :=
  match stx.isStrLit? with
  | some str => pure $ mkAtomFrom stx str
  | none     => Macro.throwUnsupported

/- Convert `notation` command lhs item into a pattern element -/
def expandNotationItemIntoPattern (stx : Syntax) : CommandElabM Syntax :=
  let k := stx.getKind
  if k == `Lean.Parser.Command.identPrec then
    mkAntiquotNode stx[0]
  else if k == strLitKind then
    liftMacroM <| strLitToPattern stx
  else
    throwUnsupportedSyntax

/-- Try to derive a `SimpleDelab` from a notation.
    The notation must be of the form `notation ... => c var_1 ... var_n`
    where `c` is a declaration in the current scope and the `var_i` are a permutation of the LHS vars. -/
def mkSimpleDelab (attrKind : Syntax) (vars : Array Syntax) (pat qrhs : Syntax) : OptionT CommandElabM Syntax := do
  match qrhs with
  | `($c:ident $args*) =>
    let [(c, [])] ← resolveGlobalName c.getId | failure
    guard <| args.all (Syntax.isIdent ∘ getAntiquotTerm)
    guard <| args.allDiff
    -- replace head constant with (unused) antiquotation so we're not dependent on the exact pretty printing of the head
    let qrhs ← `($(mkAntiquotNode (← `(_))) $args*)
    `(@[$attrKind:attrKind appUnexpander $(mkIdent c):ident] def unexpand : Lean.PrettyPrinter.Unexpander := fun
       | `($qrhs) => `($pat)
       | _        => throw ())
  | `($c:ident)       =>
    let [(c, [])] ← resolveGlobalName c.getId | failure
    `(@[$attrKind:attrKind appUnexpander $(mkIdent c):ident] def unexpand : Lean.PrettyPrinter.Unexpander := fun _ => `($pat))
  | _                 => failure

private def expandNotationAux (ref : Syntax)
    (currNamespace : Name) (attrKind : Syntax) (prec? : Option Syntax) (name? : Option Syntax) (prio? : Option Syntax) (items : Array S
  let prio ← liftMacroM <| evalOptPrio prio?
  -- build parser
  let syntaxParts ← items.mapM expandNotationItemIntoSyntaxItem
  let cat := mkIdentFrom ref `term
  let name ←
    match name? with
    | some name => pure name.getId
    | none => mkNameFromParserSyntax `term (mkNullNode syntaxParts)
```

```
  -- build macro rules
  let vars := items.filter fun item => item.getKind == `Lean.Parser.Command.identPrec
  let vars := vars.map fun var => var[0]
  let qrhs := antiquote vars rhs
  let patArgs ← items.mapM expandNotationItemIntoPattern
  /- The command `syntax [<kind>] ...` adds the current namespace to the syntax node kind.
     So, we must include current namespace when we create a pattern for the following `macro_rules` commands. -/
  let fullName := currNamespace ++ name
  let pat := Syntax.node fullName patArgs
  let stxDecl ← `($attrKind:attrKind syntax $[: $prec?]? (name := $(mkIdent name)) (priority := $(quote prio):numLit) $[$syntaxParts]*
  let macroDecl ← `(macro_rules | `($pat) => `($qrhs))
  match (← mkSimpleDelab attrKind vars pat qrhs |>.run) with
  | some delabDecl => mkNullNode #[stxDecl, macroDecl, delabDecl]
  | none           => mkNullNode #[stxDecl, macroDecl]

@[builtinCommandElab «notation»] def expandNotation : CommandElab :=
  adaptExpander fun stx => do
    let currNamespace ← getCurrNamespace
    match stx with
    | `($attrKind:attrKind notation $[: $prec? ]? $[(name := $name?)]? $[(priority := $prio?)]? $items* => $rhs) =>
      -- trigger scoped checks early and only once
      let _ ← toAttributeKind attrKind
      expandNotationAux stx currNamespace attrKind prec? name? prio? items rhs
    | _ => throwUnsupportedSyntax

/- Convert `macro` argument into a `syntax` command item -/
def expandMacroArgIntoSyntaxItem : Macro
  | `(macroArg|$id:ident:$stx)    => stx
  -- can't match against `$s:strLit%$id` because the latter part would be interpreted as an antiquotation on the token
  -- `strLit`.
  | `(macroArg|$s:macroArgSymbol) => `(stx|$(s[0]):strLit)
  | _                             => Macro.throwUnsupported

/- Convert `macro` arg into a pattern element -/
def expandMacroArgIntoPattern (stx : Syntax) : MacroM Syntax := do
  match (← expandMacros stx) with
  | `(macroArg|$id:ident:optional($stx)) =>
    mkSplicePat `optional id "?"
  | `(macroArg|$id:ident:many($stx)) =>
    mkSplicePat `many id "*"
  | `(macroArg|$id:ident:many1($stx)) =>
    mkSplicePat `many id "*"
  | `(macroArg|$id:ident:sepBy($stx, $sep:strLit $[, $stxsep]? $[, allowTrailingSep]?)) =>
    mkSplicePat `sepBy id ((isStrLit? sep).get! ++ "*")
  | `(macroArg|$id:ident:sepBy1($stx, $sep:strLit $[, $stxsep]? $[, allowTrailingSep]?)) =>
    mkSplicePat `sepBy id ((isStrLit? sep).get! ++ "*")
  | `(macroArg|$id:ident:$stx) => mkAntiquotNode id
```

```
  | `(macroArg|$s:strLit) => strLitToPattern s
  -- `"tk"%id` ~> `"tk"%$id`
  | `(macroArg|$s:macroArgSymbol) => mkNode `token_antiquot #[← strLitToPattern s[0], mkAtom "%", mkAtom "$", s[1][1]]
  | _                          => Macro.throwUnsupported
  where mkSplicePat kind id suffix :=
    mkNullNode #[mkAntiquotSuffixSpliceNode kind (mkAntiquotNode id) suffix]


/- «macro» := leading_parser suppressInsideQuot (Term.attrKind >> "macro " >> optPrecedence >> optNamedName >> optNamedPrio >> macroHea
def expandMacro (currNamespace : Name) (stx : Syntax) : CommandElabM Syntax := do
  let attrKind := stx[0]
  let prec := stx[2].getOptional?
  let name? ← liftMacroM <| expandOptNamedName stx[3]
  let prio  ← liftMacroM <| expandOptNamedPrio stx[4]
  let head := stx[5]
  let args := stx[6].getArgs
  let cat  := stx[8]
  -- build parser
  let stxPart  ← liftMacroM <| expandMacroArgIntoSyntaxItem head
  let stxParts ← liftMacroM <| args.mapM expandMacroArgIntoSyntaxItem
  let stxParts := #[stxPart] ++ stxParts
  -- name
  let name ← match name? with
    | some name => pure name
    | none => mkNameFromParserSyntax cat.getId (mkNullNode stxParts)
  -- build macro rules
  let patHead ← liftMacroM <| expandMacroArgIntoPattern head
  let patArgs ← liftMacroM <| args.mapM expandMacroArgIntoPattern
  /- The command `syntax [<kind>] ...` adds the current namespace to the syntax node kind.
     So, we must include current namespace when we create a pattern for the following `macro_rules` commands. -/
  let pat := Syntax.node (currNamespace ++ name) (#[patHead] ++ patArgs)
  if stx.getArgs.size == 11 then
    -- `stx` is of the form `macro $head $args* : $cat => term`
    let rhs := stx[10]
    let stxCmd ← `(Parser.Command.syntax| $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quo
    let macroRulesCmd ← `(macro_rules | `($pat) => $rhs)
    return mkNullNode #[stxCmd, macroRulesCmd]
  else
    -- `stx` is of the form `macro $head $args* : $cat => `( $body )`
    let rhsBody := stx[11]
    let stxCmd ← `(Parser.Command.syntax| $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quo
    let macroRulesCmd ← `(macro_rules | `($pat) => `($rhsBody))
    return mkNullNode #[stxCmd, macroRulesCmd]

@[builtinCommandElab «macro»] def elabMacro : CommandElab :=
  adaptExpander fun stx => do
    expandMacro (← getCurrNamespace) stx
```

```
builtin_initialize
  registerTraceClass `Elab.syntax

@[inline] def withExpectedType (expectedType? : Option Expr) (x : Expr → TermElabM Expr) : TermElabM Expr := do
  Term.tryPostponeIfNoneOrMVar expectedType?
  let some expectedType ← pure expectedType?
    | throwError "expected type must be known"
  x expectedType

/-
def elabTail := try (" : " >> ident) >> darrow >> termParser
def «elab» := leading_parser suppressInsideQuot (Term.attrKind >> "elab " >> optPrecedence >> optNamedName >> optNamedPrio >> elabHead
-/
def expandElab (currNamespace : Name) (stx : Syntax) : CommandElabM Syntax := do
  let ref := stx
  let attrKind := stx[0]
  let prec     := stx[2].getOptional?
  let name?   ← liftMacroM <| expandOptNamedName stx[3]
  let prio    ← liftMacroM <| expandOptNamedPrio stx[4]
  let head    := stx[5]
  let args    := stx[6].getArgs
  let cat     := stx[8]
  let expectedTypeSpec := stx[9]
  let rhs     := stx[11]
  let catName := cat.getId
  -- build parser
  let stxPart  ← liftMacroM <| expandMacroArgIntoSyntaxItem head
  let stxParts ← liftMacroM <| args.mapM expandMacroArgIntoSyntaxItem
  let stxParts := #[stxPart] ++ stxParts
  -- name
  let name ← match name? with
    | some name => pure name
    | none => mkNameFromParserSyntax cat.getId (mkNullNode stxParts)
  -- build pattern for `martch_syntax
  let patHead ← liftMacroM <| expandMacroArgIntoPattern head
  let patArgs ← liftMacroM <| args.mapM expandMacroArgIntoPattern
  let pat := Syntax.node (currNamespace ++ name) (#[patHead] ++ patArgs)
  let stxCmd ← `(Parser.Command.syntax|
    $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quote prio):numLit) $[$stxParts]* : $cat)
  let elabCmd ←
    if expectedTypeSpec.hasArgs then
      if catName == `term then
        let expId := expectedTypeSpec[1]
        `(@[termElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Term.TermElab :=
          fun stx expectedType? => match stx with
            | `($pat) => Lean.Elab.Command.withExpectedType expectedType? fun $expId => $rhs
```

```
          | _ => throwUnsupportedSyntax)
    else
      throwErrorAt expectedTypeSpec "syntax category '{catName}' does not support expected type specification"
  else if catName == `term then
    `(@[termElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Term.TermElab :=
      fun stx _ => match stx with
        | `($pat) => $rhs
        | _ => throwUnsupportedSyntax)
  else if catName == `command then
    `(@[commandElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Command.CommandElab :=
      fun
        | `($pat) => $rhs
        | _ => throwUnsupportedSyntax)
  else if catName == `tactic then
    `(@[tactic $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Tactic.Tactic :=
      fun
        | `(tactic|$pat) => $rhs
        | _ => throwUnsupportedSyntax)
  else
    -- We considered making the command extensible and support new user-defined categories. We think it is unnecessary.
    -- If users want this feature, they add their own `elab` macro that uses this one as a fallback.
    throwError "unsupported syntax category '{catName}'"
 return mkNullNode #[stxCmd, elabCmd]

@[builtinCommandElab «elab»] def elabElab : CommandElab :=
 adaptExpander fun stx => do
  expandElab (← getCurrNamespace) stx

end Lean.Elab.Command
::::::::::::::
Elab/SyntheticMVars.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Util.ForEachExpr
import Lean.Elab.Term
import Lean.Elab.Tactic.Basic

namespace Lean.Elab.Term
open Tactic (TacticM evalTactic getUnsolvedGoals)
open Meta

/-- Auxiliary function used to implement `synthesizeSyntheticMVars`. -/
private def resumeElabTerm (stx : Syntax) (expectedType? : Option Expr) (errToSorry := true) : TermElabM Expr :=
```

```
  -- Remark: if `ctx.errToSorry` is already false, then we don't enable it. Recall tactics disable `errToSorry`
  withReader (fun ctx => { ctx with errToSorry := ctx.errToSorry && errToSorry }) do
    elabTerm stx expectedType? false

/--
  Try to elaborate `stx` that was postponed by an elaboration method using `Expection.postpone`.
  It returns `true` if it succeeded, and `false` otherwise.
  It is used to implement `synthesizeSyntheticMVars`. -/
private def resumePostponed (savedContext : SavedContext) (stx : Syntax) (mvarId : MVarId) (postponeOnError : Bool) : TermElabM Bool :=
  withRef stx <| withMVarContext mvarId do
    let s ← get
    try
      withSavedContext savedContext do
        let mvarDecl     ← getMVarDecl mvarId
        let expectedType ← instantiateMVars mvarDecl.type
        withInfoHole mvarId do
          let result ← resumeElabTerm stx expectedType (!postponeOnError)
          /- We must ensure `result` has the expected type because it is the one expected by the method that postponed stx.
            That is, the method does not have an opportunity to check whether `result` has the expected type or not. -/
          let result ← withRef stx <| ensureHasType expectedType result
          /- We must perform `occursCheck` here since `result` may contain `mvarId` when it has synthetic `sorry`s. -/
          if (← occursCheck mvarId result) then
            assignExprMVar mvarId result
            return true
          else
            return false
    catch
     | ex@(Exception.internal id _) =>
       if id == postponeExceptionId then
         set s
         return false
       else
         throw ex
     | ex@(Exception.error _ _) =>
       if postponeOnError then
         set s
         return false
       else
         logException ex
         return true

/--
  Similar to `synthesizeInstMVarCore`, but makes sure that `instMVar` local context and instances
  are used. It also logs any error message produced. -/
private def synthesizePendingInstMVar (instMVar : MVarId) : TermElabM Bool :=
  withMVarContext instMVar do
    try
```

```
      synthesizeInstMVarCore instMVar
    catch
      | ex@(Exception.error _ _) => logException ex; return true
      | _                        => unreachable!

/--
  Similar to `synthesizePendingInstMVar`, but generates type mismatch error message.
  Remark: `eNew` is of the form `@coe ... mvar`, where `mvar` is the metavariable for the `CoeT ...` instance.
  If `mvar` can be synthesized, then assign `auxMVarId := (expandCoe eNew)`.
-/
private def synthesizePendingCoeInstMVar
    (auxMVarId : MVarId) (errorMsgHeader? : Option String) (eNew : Expr) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option E
  let instMVarId := eNew.appArg!.mvarId!
  withMVarContext instMVarId do
    if (← isDefEq expectedType eType) then
      /- This case may seem counterintuitive since we created the coercion
         because the `isDefEq expectedType eType` test failed before.
         However, it may succeed here because we have more information, for example, metavariables
         occurring at `expectedType` and `eType` may have been assigned. -/
      if (← occursCheck auxMVarId e) then
        assignExprMVar auxMVarId e
        return true
      else
        return false
    try
      if (← synthesizeCoeInstMVarCore instMVarId) then
        let eNew ← expandCoe eNew
        if (← occursCheck auxMVarId eNew) then
          assignExprMVar auxMVarId eNew
          return true
      return false
    catch
      | Exception.error _ msg => throwTypeMismatchError errorMsgHeader? expectedType eType e f? msg
      | _                     => unreachable!

/--
  Try to synthesize a value for `mvarId` using the given default instance.
  Return `some (val, mvarDecls)` if successful, where `val` is the value assigned to `mvarId`, and `mvarDecls` is a list of new type cl
-/
private def tryToSynthesizeUsingDefaultInstance (mvarId : MVarId) (defaultInstance : Name) : TermElabM (Option (Expr × List SyntheticMV
  commitWhenSome? do
    let candidate ← mkConstWithFreshMVarLevels defaultInstance
    let (mvars, bis, _) ← forallMetaTelescopeReducing (← inferType candidate)
    let candidate := mkAppN candidate mvars
    trace[Elab.resume] "trying default instance for {mkMVar mvarId} := {candidate}"
    if (← isDefEqGuarded (mkMVar mvarId) candidate) then
      -- Succeeded. Collect new TC problems
```

```
      let mut result := []
      for i in [:bis.size] do
        if bis[i] == BinderInfo.instImplicit then
           result := { mvarId := mvars[i].mvarId!, stx := (← getRef), kind := SyntheticMVarKind.typeClass } :: result
      trace[Elab.resume] "worked"
      return some (candidate, result)
    else
      return none

private def tryToSynthesizeUsingDefaultInstances (mvarId : MVarId) (prio : Nat) : TermElabM (Option (Expr × List SyntheticMVarDecl)) :=
  withMVarContext mvarId do
    let mvarType := (← Meta.getMVarDecl mvarId).type
    match (← isClass? mvarType) with
    | none => return none
    | some className =>
      match (← getDefaultInstances className) with
      | [] => return none
      | defaultInstances =>
        for (defaultInstance, instPrio) in defaultInstances do
          if instPrio == prio then
            match (← tryToSynthesizeUsingDefaultInstance mvarId defaultInstance) with
            | some result => return some result
            | none => continue
        return none

/- Used to implement `synthesizeUsingDefault`. This method only consider default instances with the given priority. -/
private def synthesizeUsingDefaultPrio (prio : Nat) : TermElabM Bool := do
  let rec visit (syntheticMVars : List SyntheticMVarDecl) (syntheticMVarsNew : List SyntheticMVarDecl) : TermElabM Bool := do
    match syntheticMVars with
    | [] => return false
    | mvarDecl :: mvarDecls =>
      match mvarDecl.kind with
      | SyntheticMVarKind.typeClass =>
        match (← withRef mvarDecl.stx <| tryToSynthesizeUsingDefaultInstances mvarDecl.mvarId prio) with
        | none => visit mvarDecls (mvarDecl :: syntheticMVarsNew)
        | some (val, newMVarDecls) =>
          for newMVarDecl in newMVarDecls do
            -- Register that `newMVarDecl.mvarId`s are implicit arguments of the value assigned to `mvarDecl.mvarId`
            registerMVarErrorImplicitArgInfo newMVarDecl.mvarId (← getRef) val
          let syntheticMVarsNew := newMVarDecls ++ syntheticMVarsNew
          let syntheticMVarsNew := mvarDecls.reverse ++ syntheticMVarsNew
          modify fun s => { s with syntheticMVars := syntheticMVarsNew }
          return true
      | _ => visit mvarDecls (mvarDecl :: syntheticMVarsNew)
  /- Recall that s.syntheticMVars is essentially a stack. The first metavariable was the last one created.
     We want to apply the default instance in reverse creation order. Otherwise,
     `toString 0` will produce a `OfNat String _` cannot be synthesized error. -/
```

```
  visit (← get).syntheticMVars.reverse []

/--
  Apply default value to any pending synthetic metavariable of kind `SyntheticMVarKind.withDefault`
  Return true if something was synthesized. -/
private def synthesizeUsingDefault : TermElabM Bool := do
  let prioSet ← getDefaultInstancesPriorities
  /- Recall that `prioSet` is stored in descending order -/
  for prio in prioSet do
    if (← synthesizeUsingDefaultPrio prio) then
      return true
  return false

/-- Report an error for each synthetic metavariable that could not be resolved. -/
private def reportStuckSyntheticMVars : TermElabM Unit := do
  let syntheticMVars ← modifyGet fun s => (s.syntheticMVars, { s with syntheticMVars := [] })
  for mvarSyntheticDecl in syntheticMVars do
    withRef mvarSyntheticDecl.stx do
    match mvarSyntheticDecl.kind with
    | SyntheticMVarKind.typeClass =>
      withMVarContext mvarSyntheticDecl.mvarId do
        let mvarDecl ← getMVarDecl mvarSyntheticDecl.mvarId
        unless (← get).messages.hasErrors do
          throwError "typeclass instance problem is stuck, it is often due to metavariables{indentExpr mvarDecl.type}"
    | SyntheticMVarKind.coe header eNew expectedType eType e f? =>
      let mvarId := eNew.appArg!.mvarId!
      withMVarContext mvarId do
        let mvarDecl ← getMVarDecl mvarId
        throwTypeMismatchError header expectedType eType e f? (some ("failed to create type class instance for " ++ indentExpr mvarDecl
    | _ => unreachable! -- TODO handle other cases.

private def getSomeSynthethicMVarsRef : TermElabM Syntax := do
  let s ← get
  match s.syntheticMVars.find? fun (mvarDecl : SyntheticMVarDecl) => !mvarDecl.stx.getPos?.isNone with
  | some mvarDecl => return mvarDecl.stx
  | none          => return Syntax.missing

mutual

  partial def liftTacticElabM {α} (mvarId : MVarId) (x : TacticM α) : TermElabM α :=
    withMVarContext mvarId do
      let savedSyntheticMVars := (← get).syntheticMVars
      modify fun s => { s with syntheticMVars := [] }
      try
        let a ← x.run' { main := mvarId } { goals := [mvarId] }
        synthesizeSyntheticMVars (mayPostpone := false)
        pure a
```

```
    finally
      modify fun s => { s with syntheticMVars := savedSyntheticMVars }

partial def runTactic (mvarId : MVarId) (tacticCode : Syntax) : TermElabM Unit := do
  /- Recall, `tacticCode` is the whole `by ...` expression.
     We store the `by` because in the future we want to save the initial state information at the `by` position. -/
  let code := tacticCode[1]
  modifyThe Meta.State fun s => { s with mctx := s.mctx.instantiateMVarDeclMVars mvarId }
  let remainingGoals ← withInfoHole mvarId do liftTacticElabM mvarId do evalTactic code; getUnsolvedGoals
  unless remainingGoals.isEmpty do reportUnsolvedGoals remainingGoals

/-- Try to synthesize the given pending synthetic metavariable. -/
private partial def synthesizeSyntheticMVar (mvarSyntheticDecl : SyntheticMVarDecl) (postponeOnError : Bool) (runTactics : Bool) : Te
  withRef mvarSyntheticDecl.stx do
  match mvarSyntheticDecl.kind with
  | SyntheticMVarKind.typeClass => synthesizePendingInstMVar mvarSyntheticDecl.mvarId
  | SyntheticMVarKind.coe header? eNew expectedType eType e f? => synthesizePendingCoeInstMVar mvarSyntheticDecl.mvarId header? eNew 
  -- NOTE: actual processing at `synthesizeSyntheticMVarsAux`
  | SyntheticMVarKind.postponed savedContext => resumePostponed savedContext mvarSyntheticDecl.stx mvarSyntheticDecl.mvarId postponeO
  | SyntheticMVarKind.tactic tacticCode savedContext =>
    withSavedContext savedContext do
      if runTactics then
        runTactic mvarSyntheticDecl.mvarId tacticCode
        return true
      else
        return false
/--
  Try to synthesize the current list of pending synthetic metavariables.
  Return `true` if at least one of them was synthesized. -/
private partial def synthesizeSyntheticMVarsStep (postponeOnError : Bool) (runTactics : Bool) : TermElabM Bool := do
  let ctx ← read
  traceAtCmdPos `Elab.resuming fun _ =>
    m!"resuming synthetic metavariables, mayPostpone: {ctx.mayPostpone}, postponeOnError: {postponeOnError}"
  let syntheticMVars    := (← get).syntheticMVars
  let numSyntheticMVars := syntheticMVars.length
  -- We reset `syntheticMVars` because new synthetic metavariables may be created by `synthesizeSyntheticMVar`.
  modify fun s => { s with syntheticMVars := [] }
  -- Recall that `syntheticMVars` is a list where head is the most recent pending synthetic metavariable.
  -- We use `filterRevM` instead of `filterM` to make sure we process the synthetic metavariables using the order they were created.
  -- It would not be incorrect to use `filterM`.
  let remainingSyntheticMVars ← syntheticMVars.filterRevM fun mvarDecl => do
     -- We use `traceM` because we want to make sure the metavar local context is used to trace the message
     traceM `Elab.postpone (withMVarContext mvarDecl.mvarId do addMessageContext m!"resuming {mkMVar mvarDecl.mvarId}")
     let succeeded ← synthesizeSyntheticMVar mvarDecl postponeOnError runTactics
     trace[Elab.postpone] if succeeded then fmt "succeeded" else fmt "not ready yet"
     pure !succeeded
  -- Merge new synthetic metavariables with `remainingSyntheticMVars`, i.e., metavariables that still couldn't be synthesized
```

```
    modify fun s => { s with syntheticMVars := s.syntheticMVars ++ remainingSyntheticMVars }
    return numSyntheticMVars != remainingSyntheticMVars.length

  /--
    Try to process pending synthetic metavariables. If `mayPostpone == false`,
    then `syntheticMVars` is `[]` after executing this method.

    It keeps executing `synthesizeSyntheticMVarsStep` while progress is being made.
    If `mayPostpone == false`, then it applies default instances to `SyntheticMVarKind.typeClass` (if available)
    metavariables that are still unresolved, and then tries to resolve metavariables
    with `mayPostpone == false`. That is, we force them to produce error messages and/or commit to
    a "best option". If, after that, we still haven't made progress, we report "stuck" errors. -/
  partial def synthesizeSyntheticMVars (mayPostpone := true) : TermElabM Unit :=
    let rec loop (u : Unit) : TermElabM Unit := do
      withRef (← getSomeSynthethicMVarsRef) <| withIncRecDepth do
        unless (← get).syntheticMVars.isEmpty do
          if ← synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := false) then
            loop ()
          else if !mayPostpone then
            /- Resume pending metavariables with "elaboration postponement" disabled.
               We postpone elaboration errors in this step by setting `postponeOnError := true`.
               Example:
               ```
               #check let x := ⟨1, 2⟩; Prod.fst x
               ```
               The term `⟨1, 2⟩` can't be elaborated because the expected type is not know.
               The `x` at `Prod.fst x` is not elaborated because the type of `x` is not known.
               When we execute the following step with "elaboration postponement" disabled,
               the elaborator fails at `⟨1, 2⟩` and postpones it, and succeeds at `x` and learns
               that its type must be of the form `Prod ?α ?β`.

               Recall that we postponed `x` at `Prod.fst x` because its type it is not known.
               We the type of `x` may learn later its type and it may contain implicit and/or auto arguments.
               By disabling postponement, we are essentially giving up the opportunity of learning `x`s type
               and assume it does not have implict and/or auto arguments. -/
            if ← withoutPostponing <| synthesizeSyntheticMVarsStep (postponeOnError := true) (runTactics := false) then
              loop ()
            else if ← synthesizeUsingDefault then
              loop ()
            else if ← withoutPostponing <| synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := false) then
              loop ()
            else if ← synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := true) then
              loop ()
            else
              reportStuckSyntheticMVars
    loop ()
end
```

```lean

def synthesizeSyntheticMVarsNoPostponing : TermElabM Unit :=
  synthesizeSyntheticMVars (mayPostpone := false)

/- Keep invoking `synthesizeUsingDefault` until it returns false. -/
private partial def synthesizeUsingDefaultLoop : TermElabM Unit := do
  if (← synthesizeUsingDefault) then
    synthesizeSyntheticMVars (mayPostpone := true)
    synthesizeUsingDefaultLoop

def synthesizeSyntheticMVarsUsingDefault : TermElabM Unit := do
  synthesizeSyntheticMVars (mayPostpone := true)
  synthesizeUsingDefaultLoop

private partial def withSynthesizeImp {α} (k : TermElabM α) (mayPostpone : Bool) : TermElabM α := do
  let syntheticMVarsSaved := (← get).syntheticMVars
  modify fun s => { s with syntheticMVars := [] }
  try
    let a ← k
    synthesizeSyntheticMVars mayPostpone
    if mayPostpone then
      synthesizeUsingDefaultLoop
    return a
  finally
    modify fun s => { s with syntheticMVars := s.syntheticMVars ++ syntheticMVarsSaved }

/--
  Execute `k`, and synthesize pending synthetic metavariables created while executing `k` are solved.
  If `mayPostpone == false`, then all of them must be synthesized.
  Remark: even if `mayPostpone == true`, the method still uses `synthesizeUsingDefault` -/
@[inline] def withSynthesize [MonadFunctorT TermElabM m] [Monad m] (k : m α) (mayPostpone := false) : m α :=
  monadMap (m := TermElabM) (withSynthesizeImp . mayPostpone) k

/-- Elaborate `stx`, and make sure all pending synthetic metavariables created while elaborating `stx` are solved. -/
def elabTermAndSynthesize (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr :=
  withRef stx do
    instantiateMVars (← withSynthesize <| elabTerm stx expectedType?)

end Lean.Elab.Term
::::::::::::::
Elab/Tactic.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
```

```lean
import Lean.Elab.Term
import Lean.Elab.Tactic.Basic
import Lean.Elab.Tactic.ElabTerm
import Lean.Elab.Tactic.Induction
import Lean.Elab.Tactic.Generalize
import Lean.Elab.Tactic.Injection
import Lean.Elab.Tactic.Match
import Lean.Elab.Tactic.Rewrite
import Lean.Elab.Tactic.Location
import Lean.Elab.Tactic.Simp
::::::::::::::
Elab/Term.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.ResolveName
import Lean.Util.Sorry
import Lean.Util.ReplaceExpr
import Lean.Structure
import Lean.Meta.ExprDefEq
import Lean.Meta.AppBuilder
import Lean.Meta.SynthInstance
import Lean.Meta.CollectMVars
import Lean.Meta.Coe
import Lean.Meta.Tactic.Util
import Lean.Hygiene
import Lean.Util.RecDepth
import Lean.Elab.Log
import Lean.Elab.Level
import Lean.Elab.Attributes
import Lean.Elab.AutoBound
import Lean.Elab.InfoTree
import Lean.Elab.Open
import Lean.Elab.SetOption

namespace Lean.Elab.Term
/-
  Set isDefEq configuration for the elaborator.
  Note that we enable all approximations but `quasiPatternApprox`

  In Lean3 and Lean 4, we used to use the quasi-pattern approximation during elaboration.
  The example:
  ```
  def ex : StateT δ (StateT σ Id) σ :=
```
```

```
  monadLift (get : StateT σ Id σ)
  ```
  demonstrates why it produces counterintuitive behavior.
  We have the `Monad-lift` application:
  ```
  @monadLift ?m ?n ?c ?α (get : StateT σ id σ) : ?n ?α
  ```
  It produces the following unification problem when we process the expected type:
  ```
  ?n ?α =?= StateT δ (StateT σ id) σ
  ==> (approximate using first-order unification)
  ?n := StateT δ (StateT σ id)
  ?α := σ
  ```
  Then, we need to solve:
  ```
  ?m ?α =?= StateT σ id σ
  ==> instantiate metavars
  ?m σ =?= StateT σ id σ
  ==> (approximate since it is a quasi-pattern unification constraint)
  ?m := fun σ => StateT σ id σ
  ```
  Note that the constraint is not a Milner pattern because σ is in
  the local context of `?m`. We are ignoring the other possible solutions:
  ```
  ?m := fun σ' => StateT σ id σ
  ?m := fun σ' => StateT σ' id σ
  ?m := fun σ' => StateT σ id σ'
  ```

  We need the quasi-pattern approximation for elaborating recursor-like expressions (e.g., dependent `match with` expressions).

  If we had use first-order unification, then we would have produced
  the right answer: `?m := StateT σ id`

  Haskell would work on this example since it always uses
  first-order unification.
-/
def setElabConfig (cfg : Meta.Config) : Meta.Config :=
  { cfg with foApprox := true, ctxApprox := true, constApprox := false, quasiPatternApprox := false }

structure Context where
  fileName   : String
  fileMap   : FileMap
  declName?   : Option Name   := none
  macroStack   : MacroStack   := []
  currMacroScope : MacroScope   := firstFrontendMacroScope
```
```

```
  /- When `mayPostpone == true`, an elaboration function may interrupt its execution by throwing `Exception.postpone`.
     The function `elabTerm` catches this exception and creates fresh synthetic metavariable `?m`, stores `?m` in
     the list of pending synthetic metavariables, and returns `?m`. -/
  mayPostpone     : Bool           := true
  /- When `errToSorry` is set to true, the method `elabTerm` catches
     exceptions and converts them into synthetic `sorry`s.
     The implementation of choice nodes and overloaded symbols rely on the fact
     that when `errToSorry` is set to false for an elaboration function `F`, then
     `errToSorry` remains `false` for all elaboration functions invoked by `F`.
     That is, it is safe to transition `errToSorry` from `true` to `false`, but
     we must not set `errToSorry` to `true` when it is currently set to `false`. -/
  errToSorry      : Bool           := true
  /- When `autoBoundImplicit` is set to true, instead of producing
     an "unknown identifier" error for unbound variables, we generate an
     internal exception. This exception is caught at `elabBinders` and
     `elabTypeWithUnboldImplicit`. Both methods add implicit declarations
     for the unbound variable and try again. -/
  autoBoundImplicit  : Bool            := false
  autoBoundImplicits : Std.PArray Expr := {}
  /-- Map from user name to internal unique name -/
  sectionVars        : NameMap Name    := {}
  /-- Map from internal name to fvar -/
  sectionFVars       : NameMap Expr    := {}
  /-- Enable/disable implicit lambdas feature. -/
  implicitLambda     : Bool            := true

/-- Saved context for postponed terms and tactics to be executed. -/
structure SavedContext where
  declName?  : Option Name
  options    : Options
  openDecls  : List OpenDecl
  macroStack : MacroStack
  errToSorry : Bool

/-- We use synthetic metavariables as placeholders for pending elaboration steps. -/
inductive SyntheticMVarKind where
  -- typeclass instance search
  | typeClass
  /- Similar to typeClass, but error messages are different.
     if `f?` is `some f`, we produce an application type mismatch error message.
     Otherwise, if `header?` is `some header`, we generate the error `(header ++ "has type" ++ eType ++ "but it is expected to have typ
     Otherwise, we generate the error `("type mismatch" ++ e ++ "has type" ++ eType ++ "but it is expected to have type" ++ expectedTyp
  | coe (header? : Option String) (eNew : Expr) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr)
  -- tactic block execution
  | tactic (tacticCode : Syntax) (ctx : SavedContext)
  -- `elabTerm` call that threw `Exception.postpone` (input is stored at `SyntheticMVarDecl.ref`)
  | postponed (ctx : SavedContext)
```

```
instance : ToString SyntheticMVarKind where
  toString
    | SyntheticMVarKind.typeClass    => "typeclass"
    | SyntheticMVarKind.coe ..       => "coe"
    | SyntheticMVarKind.tactic ..    => "tactic"
    | SyntheticMVarKind.postponed .. => "postponed"

structure SyntheticMVarDecl where
  mvarId : MVarId
  stx : Syntax
  kind : SyntheticMVarKind

inductive MVarErrorKind where
  | implicitArg (ctx : Expr)
  | hole
  | custom (msgData : MessageData)

instance : ToString MVarErrorKind where
  toString
    | MVarErrorKind.implicitArg ctx => "implicitArg"
    | MVarErrorKind.hole            => "hole"
    | MVarErrorKind.custom msg      => "custom"

structure MVarErrorInfo where
  mvarId    : MVarId
  ref       : Syntax
  kind      : MVarErrorKind

structure LetRecToLift where
  ref            : Syntax
  fvarId         : FVarId
  attrs          : Array Attribute
  shortDeclName  : Name
  declName       : Name
  lctx           : LocalContext
  localInstances : LocalInstances
  type           : Expr
  val            : Expr
  mvarId         : MVarId

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

```
  infoState         : InfoState := {}
  deriving Inhabited

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

open Meta

instance : Inhabited (TermElabM α) where
  default := throw arbitrary

structure SavedState where
  meta   : Meta.SavedState
  «elab» : State
  deriving Inhabited

protected def saveState : TermElabM SavedState := do
  pure { meta := (← Meta.saveState), «elab» := (← get) }

def SavedState.restore (s : SavedState) : TermElabM Unit := do
  let traceState ← getTraceState -- We never backtrack trace message
  -- We also preserve `TacticInfo` nodes to be able to display the tactic state of broken tactic scripts
  let infoStateNew := (← get).infoState
  let oldInfoSize  := s.elab.infoState.trees.size
  s.meta.restore
  set s.elab
  setTraceState traceState
  -- Add new `TacticInfo` nodes back to restored `infoState`
  modify fun s => { s with
    infoState.trees := infoStateNew.trees.foldl (init := s.infoState.trees) (start := oldInfoSize) fun trees info =>
      match info with
      | InfoTree.node (Info.ofTacticInfo _) _ => trees.push info
      | _ => trees
  }

instance : MonadBacktrack SavedState TermElabM where
  saveState      := Term.saveState
  restoreState b := b.restore

abbrev TermElabResult (α : Type) := EStateM.Result Exception SavedState α

instance [Inhabited α] : Inhabited (TermElabResult α) where
  default := EStateM.Result.ok arbitrary arbitrary

def setMessageLog (messages : MessageLog) : TermElabM Unit :=
  modify fun s => { s with messages := messages }
```

```
def resetMessageLog : TermElabM Unit :=
  setMessageLog {}

def getMessageLog : TermElabM MessageLog :=
  return (← get).messages

/--
  Execute `x`, save resulting expression and new state.
  If `x` fails, then it also stores exception and new state.
  Remark: we do not capture `Exception.postpone`. -/
@[inline] def observing (x : TermElabM α) : TermElabM (TermElabResult α) := do
  let s ← saveState
  try
    let e ← x
    let sNew ← saveState
    s.restore
    pure (EStateM.Result.ok e sNew)
  catch
    | ex@(Exception.error _ _) =>
      let sNew ← saveState
      s.restore
      pure (EStateM.Result.error ex sNew)
    | ex@(Exception.internal id _) =>
      if id == postponeExceptionId then s.restore
      throw ex

/--
  Apply the result/exception and state captured with `observing`.
  We use this method to implement overloaded notation and symbols. -/
@[inline] def applyResult (result : TermElabResult α) : TermElabM α :=
  match result with
  | EStateM.Result.ok a r     => do r.restore; pure a
  | EStateM.Result.error ex r => do r.restore; throw ex

/--
  Execute `x`, but keep state modifications only if `x` did not postpone.
  This method is useful to implement elaboration functions that cannot decide whether
  they need to postpone or not without updating the state. -/
def commitIfDidNotPostpone (x : TermElabM α) : TermElabM α := do
  -- We just reuse the implementation of `observing` and `applyResult`.
  let r ← observing x
  applyResult r

/--
  Execute `x` but discard changes performed at `Term.State` and `Meta.State`.
  Recall that the environment is at `Core.State`. Thus, any updates to it will
  be preserved. This method is useful for performing computations where all
```

```
  metavariable must be resolved or discarded. -/
def withoutModifyingElabMetaState (x : TermElabM α) : TermElabM α := do
  let s ← get
  let sMeta ← getThe Meta.State
  try
    x
  finally
    set s
    set sMeta

def getLevelNames : TermElabM (List Name) :=
  return (← get).levelNames

def getFVarLocalDecl! (fvar : Expr) : TermElabM LocalDecl := do
  match (← getLCtx).find? fvar.fvarId! with
  | some d => pure d
  | none   => unreachable!

instance : AddErrorMessageContext TermElabM where
  add ref msg := do
    let ctx ← read
    let ref := getBetterRef ref ctx.macroStack
    let msg ← addMessageContext msg
    let msg ← addMacroStack msg ctx.macroStack
    pure (ref, msg)

instance : MonadLog TermElabM where
  getRef      := getRef
  getFileMap  := return (← read).fileMap
  getFileName := return (← read).fileName
  logMessage msg := do
    let ctx ← readThe Core.Context
    let msg := { msg with data := MessageData.withNamingContext { currNamespace := ctx.currNamespace, openDecls := ctx.openDecls } msg.
    modify fun s => { s with messages := s.messages.add msg }

protected def getCurrMacroScope : TermElabM MacroScope := do pure (← read).currMacroScope
protected def getMainModule     : TermElabM Name := do pure (← getEnv).mainModule

@[inline] protected def withFreshMacroScope (x : TermElabM α) : TermElabM α := do
  let fresh ← modifyGetThe Core.State (fun st => (st.nextMacroScope, { st with nextMacroScope := st.nextMacroScope + 1 }))
  withReader (fun ctx => { ctx with currMacroScope := fresh }) x

instance : MonadQuotation TermElabM where
  getCurrMacroScope   := Term.getCurrMacroScope
  getMainModule       := Term.getMainModule
  withFreshMacroScope := Term.withFreshMacroScope
```

```
instance : MonadInfoTree TermElabM where
  getInfoState      := return (← get).infoState
  modifyInfoState f := modify fun s => { s with infoState := f s.infoState }

unsafe def mkTermElabAttributeUnsafe : IO (KeyedDeclsAttribute TermElab) :=
  mkElabAttribute TermElab `Lean.Elab.Term.termElabAttribute `builtinTermElab `termElab `Lean.Parser.Term `Lean.Elab.Term.TermElab "ter

@[implementedBy mkTermElabAttributeUnsafe]
constant mkTermElabAttribute : IO (KeyedDeclsAttribute TermElab)

builtin_initialize termElabAttribute : KeyedDeclsAttribute TermElab ← mkTermElabAttribute

/--
  Auxiliary datatatype for presenting a Lean lvalue modifier.
  We represent a unelaborated lvalue as a `Syntax` (or `Expr`) and `List LVal`.
  Example: `a.foo[i].1` is represented as the `Syntax` `a` and the list
  `[LVal.fieldName "foo", LVal.getOp i, LVal.fieldIdx 1]`.
  Recall that the notation `a[i]` is not just for accessing arrays in Lean. -/
inductive LVal where
  | fieldIdx  (ref : Syntax) (i : Nat)
    /- Field `suffix?` is for producing better error messages because `x.y` may be a field access or a hierachical/composite name.
       `ref` is the syntax object representing the field. `targetStx` is the target object being accessed. -/
  | fieldName (ref : Syntax) (name : String) (suffix? : Option Name) (targetStx : Syntax)
  | getOp     (ref : Syntax) (idx : Syntax)

def LVal.getRef : LVal → Syntax
  | LVal.fieldIdx ref _    => ref
  | LVal.fieldName ref ..  => ref
  | LVal.getOp ref _       => ref

def LVal.isFieldName : LVal → Bool
  | LVal.fieldName .. => true
  | _ => false

instance : ToString LVal where
  toString
    | LVal.fieldIdx _ i     => toString i
    | LVal.fieldName _ n .. => n
    | LVal.getOp _ idx      => "[" ++ toString idx ++ "]"

def getDeclName? : TermElabM (Option Name) := return (← read).declName?
def getLetRecsToLift : TermElabM (List LetRecToLift) := return (← get).letRecsToLift
def isExprMVarAssigned (mvarId : MVarId) : TermElabM Bool := return (← getMCtx).isExprAssigned mvarId
def getMVarDecl (mvarId : MVarId) : TermElabM MetavarDecl := return (← getMCtx).getDecl mvarId
def assignLevelMVar (mvarId : MVarId) (val : Level) : TermElabM Unit := modifyThe Meta.State fun s => { s with mctx := s.mctx.assignLev

def withDeclName (name : Name) (x : TermElabM α) : TermElabM α :=
```

```lean
  withReader (fun ctx => { ctx with declName? := name }) x

def setLevelNames (levelNames : List Name) : TermElabM Unit :=
  modify fun s => { s with levelNames := levelNames }

def withLevelNames (levelNames : List Name) (x : TermElabM α) : TermElabM α := do
  let levelNamesSaved ← getLevelNames
  setLevelNames levelNames
  try x finally setLevelNames levelNamesSaved

def withoutErrToSorry (x : TermElabM α) : TermElabM α :=
  withReader (fun ctx => { ctx with errToSorry := false }) x

/-- For testing `TermElabM` methods. The #eval command will sign the error. -/
def throwErrorIfErrors : TermElabM Unit := do
  if (← get).messages.hasErrors then
    throwError "Error(s)"

@[inline] def traceAtCmdPos (cls : Name) (msg : Unit → MessageData) : TermElabM Unit :=
withRef Syntax.missing $ trace cls msg

def ppGoal (mvarId : MVarId) : TermElabM Format :=
  Meta.ppGoal mvarId

open Level (LevelElabM)

def liftLevelM (x : LevelElabM α) : TermElabM α := do
  let ctx ← read
  let ref ← getRef
  let mctx ← getMCtx
  let ngen ← getNGen
  let lvlCtx : Level.Context := { options := (← getOptions), ref := ref, autoBoundImplicit := ctx.autoBoundImplicit }
  match (x lvlCtx).run { ngen := ngen, mctx := mctx, levelNames := (← getLevelNames) } with
  | EStateM.Result.ok a newS  => setMCtx newS.mctx; setNGen newS.ngen; setLevelNames newS.levelNames; pure a
  | EStateM.Result.error ex _ => throw ex

def elabLevel (stx : Syntax) : TermElabM Level :=
  liftLevelM $ Level.elabLevel stx

/- Elaborate `x` with `stx` on the macro stack -/
@[inline] def withMacroExpansion (beforeStx afterStx : Syntax) (x : TermElabM α) : TermElabM α :=
  withMacroExpansionInfo beforeStx afterStx do
    withReader (fun ctx => { ctx with macroStack := { before := beforeStx, after := afterStx } :: ctx.macroStack }) x

/-
  Add the given metavariable to the list of pending synthetic metavariables.
  The method `synthesizeSyntheticMVars` is used to process the metavariables on this list. -/
```

```
def registerSyntheticMVar (stx : Syntax) (mvarId : MVarId) (kind : SyntheticMVarKind) : TermElabM Unit := do
  modify fun s => { s with syntheticMVars := { mvarId := mvarId, stx := stx, kind := kind } :: s.syntheticMVars }

def registerSyntheticMVarWithCurrRef (mvarId : MVarId) (kind : SyntheticMVarKind) : TermElabM Unit := do
  registerSyntheticMVar (← getRef) mvarId kind

def registerMVarErrorHoleInfo (mvarId : MVarId) (ref : Syntax) : TermElabM Unit := do
  modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.hole } :: s.mvarErrorInfos }

def registerMVarErrorImplicitArgInfo (mvarId : MVarId) (ref : Syntax) (app : Expr) : TermElabM Unit := do
  modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.implicitArg app } :: s.mvarErrorInfo

def registerMVarErrorCustomInfo (mvarId : MVarId) (ref : Syntax) (msgData : MessageData) : TermElabM Unit := do
  modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.custom msgData } :: s.mvarErrorInfos

def registerCustomErrorIfMVar (e : Expr) (ref : Syntax) (msgData : MessageData) : TermElabM Unit :=
  match e.getAppFn with
  | Expr.mvar mvarId _ => registerMVarErrorCustomInfo mvarId ref msgData
  | _ => pure ()

/-
  Auxiliary method for reporting errors of the form "... contains metavariables ...".
  This kind of error is thrown, for example, at `Match.lean` where elaboration
  cannot continue if there are metavariables in patterns.
  We only want to log it if we haven't logged any error so far. -/
def throwMVarError (m : MessageData) : TermElabM α := do
  if (← get).messages.hasErrors then
    throwAbortTerm
  else
    throwError m

def MVarErrorInfo.logError (mvarErrorInfo : MVarErrorInfo) (extraMsg? : Option MessageData) : TermElabM Unit := do
  match mvarErrorInfo.kind with
  | MVarErrorKind.implicitArg app => do
    let app ← instantiateMVars app
    let msg : MessageData := m!"don't know how to synthesize implicit argument{indentExpr app.setAppPPExplicitForExposingMVars}"
    let msg := msg ++ Format.line ++ "context:" ++ Format.line ++ MessageData.ofGoal mvarErrorInfo.mvarId
    logErrorAt mvarErrorInfo.ref (appendExtra msg)
  | MVarErrorKind.hole => do
    let msg : MessageData := "don't know how to synthesize placeholder"
    let msg := msg ++ Format.line ++ "context:" ++ Format.line ++ MessageData.ofGoal mvarErrorInfo.mvarId
    logErrorAt mvarErrorInfo.ref (MessageData.tagged `Elab.synthPlaceholder <| appendExtra msg)
  | MVarErrorKind.custom msg =>
    logErrorAt mvarErrorInfo.ref (appendExtra msg)
where
  appendExtra (msg : MessageData) : MessageData :=
    match extraMsg? with
```

```
    | none => msg
    | some extraMsg => msg ++ extraMsg

/--
  Try to log errors for the unassigned metavariables `pendingMVarIds`.

  Return `true` if there were "unfilled holes", and we should "abort" declaration.
  TODO: try to fill "all" holes using synthetic "sorry's"

  Remark: We only log the "unfilled holes" as new errors if no error has been logged so far. -/
def logUnassignedUsingErrorInfos (pendingMVarIds : Array MVarId) (extraMsg? : Option MessageData := none) : TermElabM Bool := do
  let s ← get
  let hasOtherErrors := s.messages.hasErrors
  let mut hasNewErrors := false
  let mut alreadyVisited : NameSet := {}
  for mvarErrorInfo in s.mvarErrorInfos do
    let mvarId := mvarErrorInfo.mvarId
    unless alreadyVisited.contains mvarId do
      alreadyVisited := alreadyVisited.insert mvarId
      let foundError ← withMVarContext mvarId do
        /- The metavariable `mvarErrorInfo.mvarId` may have been assigned or
           delayed assigned to another metavariable that is unassigned. -/
        let mvarDeps ← getMVars (mkMVar mvarId)
        if mvarDeps.any pendingMVarIds.contains then do
          unless hasOtherErrors do
            mvarErrorInfo.logError extraMsg?
          pure true
        else
          pure false
      if foundError then
        hasNewErrors := true
  return hasNewErrors

/-- Ensure metavariables registered using `registerMVarErrorInfos` (and used in the given declaration) have been assigned. -/
def ensureNoUnassignedMVars (decl : Declaration) : TermElabM Unit := do
  let pendingMVarIds ← getMVarsAtDecl decl
  if (← logUnassignedUsingErrorInfos pendingMVarIds) then
    throwAbortCommand

/-
  Execute `x` without allowing it to postpone elaboration tasks.
  That is, `tryPostpone` is a noop. -/
@[inline] def withoutPostponing (x : TermElabM α) : TermElabM α :=
  withReader (fun ctx => { ctx with mayPostpone := false }) x

/-- Creates syntax for `(` <ident> `:` <type> `)` -/
def mkExplicitBinder (ident : Syntax) (type : Syntax) : Syntax :=
```

```
  mkNode ``Lean.Parser.Term.explicitBinder #[mkAtom "(", mkNullNode #[ident], mkNullNode #[mkAtom ":", type], mkNullNode, mkAtom ")"]

/--
  Convert unassigned universe level metavariables into parameters.
  The new parameter names are of the form `u_i` where `i >= nextParamIdx`.
  The method returns the updated expression and new `nextParamIdx`.

  Remark: we make sure the generated parameter names do not clash with the universes at `ctx.levelNames`. -/
def levelMVarToParam (e : Expr) (nextParamIdx : Nat := 1) : TermElabM (Expr × Nat) := do
  let mctx ← getMCtx
  let levelNames ← getLevelNames
  let r := mctx.levelMVarToParam (fun n => levelNames.elem n) e `u nextParamIdx
  setMCtx r.mctx
  pure (r.expr, r.nextParamIdx)

/-- Variant of `levelMVarToParam` where `nextParamIdx` is stored in a state monad. -/
def levelMVarToParam' (e : Expr) : StateRefT Nat TermElabM Expr := do
  let nextParamIdx ← get
  let (e, nextParamIdx) ← levelMVarToParam e nextParamIdx
  set nextParamIdx
  pure e

/--
  Auxiliary method for creating fresh binder names.
  Do not confuse with the method for creating fresh free/meta variable ids. -/
def mkFreshBinderName : TermElabM Name :=
  withFreshMacroScope $ MonadQuotation.addMacroScope `x

/--
  Auxiliary method for creating a `Syntax.ident` containing
  a fresh name. This method is intended for creating fresh binder names.
  It is just a thin layer on top of `mkFreshUserName`. -/
def mkFreshIdent (ref : Syntax) : TermElabM Syntax :=
  return mkIdentFrom ref (← mkFreshBinderName)

private def applyAttributesCore
    (declName : Name) (attrs : Array Attribute)
    (applicationTime? : Option AttributeApplicationTime) : TermElabM Unit :=
  for attr in attrs do
    let env ← getEnv
    match getAttributeImpl env attr.name with
    | Except.error errMsg => throwError errMsg
    | Except.ok attrImpl  =>
      match applicationTime? with
      | none => attrImpl.add declName attr.stx attr.kind
      | some applicationTime =>
        if applicationTime == attrImpl.applicationTime then
```

```
        attrImpl.add declName attr.stx attr.kind

/-- Apply given attributes **at** a given application time -/
def applyAttributesAt (declName : Name) (attrs : Array Attribute) (applicationTime : AttributeApplicationTime) : TermElabM Unit :=
  applyAttributesCore declName attrs applicationTime

def applyAttributes (declName : Name) (attrs : Array Attribute) : TermElabM Unit :=
  applyAttributesCore declName attrs none

def mkTypeMismatchError (header? : Option String) (e : Expr) (eType : Expr) (expectedType : Expr) : TermElabM MessageData := do
  let header : MessageData := match header? with
    | some header => m!"{header} "
    | none        => m!"type mismatch{indentExpr e}\n"
  return m!"{header}{← mkHasTypeButIsExpectedMsg eType expectedType}"

def throwTypeMismatchError (header? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr)
    (f? : Option Expr := none) (extraMsg? : Option MessageData := none) : TermElabM α := do
  /-
    We ignore `extraMsg?` for now. In all our tests, it contained no useful information. It was
    always of the form:
    ```
    failed to synthesize instance
      CoeT <eType> <e> <expectedType>
    ```
    We should revisit this decision in the future and decide whether it may contain useful information
    or not. -/
  let extraMsg := Format.nil
  /-
  let extraMsg : MessageData := match extraMsg? with
    | none          => Format.nil
    | some extraMsg => Format.line ++ extraMsg;
  -/
  match f? with
  | none   => throwError "{← mkTypeMismatchError header? e eType expectedType}{extraMsg}"
  | some f => Meta.throwAppTypeMismatch f e extraMsg

@[inline] def withoutMacroStackAtErr (x : TermElabM α) : TermElabM α :=
  withTheReader Core.Context (fun (ctx : Core.Context) => { ctx with options := pp.macroStack.set ctx.options false }) x

/- Try to synthesize metavariable using type class resolution.
   This method assumes the local context and local instances of `instMVar` coincide
   with the current local context and local instances.
   Return `true` if the instance was synthesized successfully, and `false` if
   the instance contains unassigned metavariables that are blocking the type class
   resolution procedure. Throw an exception if resolution or assignment irrevocably fails. -/
def synthesizeInstMVarCore (instMVar : MVarId) (maxResultSize? : Option Nat := none) : TermElabM Bool := do
  let instMVarDecl ← getMVarDecl instMVar
```

```
  let type := instMVarDecl.type
  let type ← instantiateMVars type
  let result ← trySynthInstance type maxResultSize?
  match result with
  | LOption.some val =>
    if (← isExprMVarAssigned instMVar) then
      let oldVal ← instantiateMVars (mkMVar instMVar)
      unless (← isDefEq oldVal val) do
        let oldValType ← inferType oldVal
        let valType ← inferType val
        unless (← isDefEq oldValType valType) do
          throwError "synthesized type class instance type is not definitionally equal to expected type, synthesized{indentExpr val}\nh
        throwError "synthesized type class instance is not definitionally equal to expression inferred by typing rules, synthesized{ind
    else
      unless (← isDefEq (mkMVar instMVar) val) do
        throwError "failed to assign synthesized type class instance{indentExpr val}"
    pure true
  | LOption.undef    => pure false -- we will try later
  | LOption.none     => throwError "failed to synthesize instance{indentExpr type}"

register_builtin_option autoLift : Bool := {
  defValue := true
  descr    := "insert monadic lifts (i.e., `liftM` and `liftCoeM`) when needed"
}

register_builtin_option maxCoeSize : Nat := {
  defValue := 16
  descr    := "maximum number of instances used to construct an automatic coercion"
}

def synthesizeCoeInstMVarCore (instMVar : MVarId) : TermElabM Bool := do
  synthesizeInstMVarCore instMVar (some (maxCoeSize.get (← getOptions)))

/-
The coercion from `α` to `Thunk α` cannot be implemented using an instance because it would
eagerly evaluate `e` -/
def tryCoeThunk? (expectedType : Expr) (eType : Expr) (e : Expr) : TermElabM (Option Expr) := do
  match expectedType with
  | Expr.app (Expr.const ``Thunk u _) arg _ =>
    if (← isDefEq eType arg) then
      pure (some (mkApp2 (mkConst ``Thunk.mk u) arg (mkSimpleThunk e)))
    else
      pure none
  | _ =>
    pure none

/--
```

```
  Try to apply coercion to make sure `e` has type `expectedType`.
  Relevant definitions:
  ```
  class CoeT (α : Sort u) (a : α) (β : Sort v)
  abbrev coe {α : Sort u} {β : Sort v} (a : α) [CoeT α a β] : β
  ```
-/
private def tryCoe (errorMsgHeader? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr) : TermElabM Exp
  if (← isDefEq expectedType eType) then
    return e
  else match (← tryCoeThunk? expectedType eType e) with
    | some r => return r
    | none   =>
      let u ← getLevel eType
      let v ← getLevel expectedType
      let coeTInstType := mkAppN (mkConst ``CoeT [u, v]) #[eType, e, expectedType]
      let mvar ← mkFreshExprMVar coeTInstType MetavarKind.synthetic
      let eNew := mkAppN (mkConst ``coe [u, v]) #[eType, expectedType, e, mvar]
      let mvarId := mvar.mvarId!
      try
        withoutMacroStackAtErr do
          if (← synthesizeCoeInstMVarCore mvarId) then
            expandCoe eNew
          else
            -- We create an auxiliary metavariable to represent the result, because we need to execute `expandCoe`
            -- after we syntheze `mvar`
            let mvarAux ← mkFreshExprMVar expectedType MetavarKind.syntheticOpaque
            registerSyntheticMVarWithCurrRef mvarAux.mvarId! (SyntheticMVarKind.coe errorMsgHeader? eNew expectedType eType e f?)
            return mvarAux
      catch
        | Exception.error _ msg => throwTypeMismatchError errorMsgHeader? expectedType eType e f? msg
        | _                     => throwTypeMismatchError errorMsgHeader? expectedType eType e f?

def isTypeApp? (type : Expr) : TermElabM (Option (Expr × Expr)) := do
  let type ← withReducible $ whnf type
  match type with
  | Expr.app m α _ => pure (some ((← instantiateMVars m), (← instantiateMVars α)))
  | _              => pure none

def synthesizeInst (type : Expr) : TermElabM Expr := do
  let type ← instantiateMVars type
  match (← trySynthInstance type) with
  | LOption.some val => pure val
  | LOption.undef    => throwError "failed to synthesize instance{indentExpr type}"
  | LOption.none     => throwError "failed to synthesize instance{indentExpr type}"

def isMonadApp (type : Expr) : TermElabM Bool := do
```

```
  let some (m, _) ← isTypeApp? type | pure false
  return (← isMonad? m) |>.isSome

/--
  Try to coerce `a : α` into `m β` by first coercing `a : α` into `β`, and then using `pure`.
  The method is only applied if `α` is not monadic (e.g., `Nat → IO Unit`), and the head symbol
  of the resulting type is not a metavariable (e.g., `?m Unit` or `Bool → ?m Nat`).

  The main limitation of the approach above is polymorphic code. As usual, coercions and polymorphism
  do not interact well. In the example above, the lift is successfully applied to `true`, `false` and `!y`
  since none of them is polymorphic
  ```
  def f (x : Bool) : IO Bool := do
  let y ← if x == 0 then IO.println "hello"; true else false;
  !y
  ```
  On the other hand, the following fails since `+` is polymorphic
  ```
  def f (x : Bool) : IO Nat := do
  IO.prinln x
  x + x -- Error: failed to synthesize `Add (IO Nat)`
  ```
-/
private def tryPureCoe? (errorMsgHeader? : Option String) (m β α a : Expr) : TermElabM (Option Expr) :=
  commitWhenSome? do
    let doIt : TermElabM (Option Expr) := do
      try
        let aNew ← tryCoe errorMsgHeader? β α a none
        let aNew ← mkPure m aNew
        pure (some aNew)
      catch _ =>
        pure none
    forallTelescope α fun _ α => do
      if (← isMonadApp α) then
        pure none
      else if !α.getAppFn.isMVar  then
        doIt
      else
        pure none

/-
Try coercions and monad lifts to make sure `e` has type `expectedType`.

If `expectedType` is of the form `n β`, we try monad lifts and other extensions.
Otherwise, we just use the basic `tryCoe`.

Extensions for monads.
```

~~~~

Given an expected type of the form `n β`, if `eType` is of the form `α`, but not `m α`

1 - Try to coerce `α` into `β`, and use `pure` to lift it to `n α`.
    It only works if `n` implements `Pure`

If `eType` is of the form `m α`. We use the following approaches.

1- Try to unify `n` and `m`. If it succeeds, then we use
   ```
   coeM {m : Type u → Type v} {α β : Type u} [∀ a, CoeT α a β] [Monad m] (x : m α) : m β
   ```
   `n` must be a `Monad` to use this one.

2- If there is monad lift from `m` to `n` and we can unify `α` and `β`, we use
  ```
  liftM : ∀ {m : Type u_1 → Type u_2} {n : Type u_1 → Type u_3} [self : MonadLiftT m n] {α : Type u_1}, m α → n α
  ```
  Note that `n` may not be a `Monad` in this case. This happens quite a bit in code such as
  ```
  def g (x : Nat) : IO Nat := do
    IO.println x
    pure x

  def f {m} [MonadLiftT IO m] : m Nat :=
    g 10

  ```

3- If there is a monad lif from `m` to `n` and a coercion from `α` to `β`, we use
  ```
  liftCoeM {m : Type u → Type v} {n : Type u → Type w} {α β : Type u} [MonadLiftT m n] [∀ a, CoeT α a β] [Monad n] (x : m α) : n β
  ```

Note that approach 3 does not subsume 1 because it is only applicable if there is a coercion from `α` to `β` for all values in `α`.
This is not the case for example for `pure $ x > 0` when the expected type is `IO Bool`. The given type is `IO Prop`, and
we only have a coercion from decidable propositions. Approach 1 works because it constructs the coercion `CoeT (m Prop) (pure $ x > 0)
using the instance `pureCoeDepProp`.

Note that, approach 2 is more powerful than `tryCoe`.
Recall that type class resolution never assigns metavariables created by other modules.
Now, consider the following scenario
```lean
def g (x : Nat) : IO Nat := ...
deg h (x : Nat) : StateT Nat IO Nat := do
v ← g x;
IO.Println v;
~~~~

```
...
```
Let's assume there is no other occurrence of `v` in `h`.
Thus, we have that the expected of `g x` is `StateT Nat IO ?α`,
and the given type is `IO Nat`. So, even if we add a coercion.
```
instance {α m n} [MonadLiftT m n] {α} : Coe (m α) (n α) := ...
```
It is not applicable because TC would have to assign `?α := Nat`.
On the other hand, TC can easily solve `[MonadLiftT IO (StateT Nat IO)]`
since this goal does not contain any metavariables. And then, we
convert `g x` into `liftM $ g x`.
-/
private def tryLiftAndCoe (errorMsgHeader? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr) : TermEl
  let expectedType ← instantiateMVars expectedType
  let eType ← instantiateMVars eType
  let throwMismatch {α} : TermElabM α := throwTypeMismatchError errorMsgHeader? expectedType eType e f?
  let tryCoeSimple : TermElabM Expr :=
    tryCoe errorMsgHeader? expectedType eType e f?
  let some (n, β) ← isTypeApp? expectedType | tryCoeSimple
  let tryPureCoeAndSimple : TermElabM Expr := do
    if autoLift.get (← getOptions) then
      match (← tryPureCoe? errorMsgHeader? n β eType e) with
      | some eNew => pure eNew
      | none      => tryCoeSimple
    else
      tryCoeSimple
  let some (m, α) ← isTypeApp? eType | tryPureCoeAndSimple
  if (← isDefEq m n) then
    let some monadInst ← isMonad? n | tryCoeSimple
    try expandCoe (← mkAppOptM ``coeM #[m, α, β, none, monadInst, e]) catch _ => throwMismatch
  else if autoLift.get (← getOptions) then
    try
      -- Construct lift from `m` to `n`
      let monadLiftType ← mkAppM ``MonadLiftT #[m, n]
      let monadLiftVal  ← synthesizeInst monadLiftType
      let u_1 ← getDecLevel α
      let u_2 ← getDecLevel eType
      let u_3 ← getDecLevel expectedType
      let eNew := mkAppN (Lean.mkConst ``liftM [u_1, u_2, u_3]) #[m, n, monadLiftVal, α, e]
      let eNewType ← inferType eNew
      if (← isDefEq expectedType eNewType) then
        return eNew -- approach 2 worked
      else
        let some monadInst ← isMonad? n | tryCoeSimple
        let u ← getLevel α
        let v ← getLevel β
```
```

```lean
      let coeTInstType := Lean.mkForall `a BinderInfo.default α $ mkAppN (mkConst ``CoeT [u, v]) #[α, mkBVar 0, β]
      let coeTInstVal ← synthesizeInst coeTInstType
      let eNew ← expandCoe (← mkAppN (Lean.mkConst ``liftCoeM [u_1, u_2, u_3]) #[m, n, α, β, monadLiftVal, coeTInstVal, monadInst, e]
      let eNewType ← inferType eNew
      unless (← isDefEq expectedType eNewType) do throwMismatch
      return eNew -- approach 3 worked
    catch _ =>
      /-
        If `m` is not a monad, then we try to use `tryPureCoe?` and then `tryCoe?`.
        Otherwise, we just try `tryCoe?`.
      -/
      match (← isMonad? m) with
      | none   => tryPureCoeAndSimple
      | some _ => tryCoeSimple
  else
    tryCoeSimple

/--
  If `expectedType?` is `some t`, then ensure `t` and `eType` are definitionally equal.
  If they are not, then try coercions.

  Argument `f?` is used only for generating error messages. -/
def ensureHasTypeAux (expectedType? : Option Expr) (eType : Expr) (e : Expr)
    (f? : Option Expr := none) (errorMsgHeader? : Option String := none) : TermElabM Expr := do
  match expectedType? with
  | none              => pure e
  | some expectedType =>
    if (← isDefEq eType expectedType) then
      pure e
    else
      tryLiftAndCoe errorMsgHeader? expectedType eType e f?

/--
  If `expectedType?` is `some t`, then ensure `t` and type of `e` are definitionally equal.
  If they are not, then try coercions. -/
def ensureHasType (expectedType? : Option Expr) (e : Expr) (errorMsgHeader? : Option String := none) : TermElabM Expr :=
  match expectedType? with
  | none => pure e
  | _    => do
    let eType ← inferType e
    ensureHasTypeAux expectedType? eType e none errorMsgHeader?

private def mkSyntheticSorryFor (expectedType? : Option Expr) : TermElabM Expr := do
  let expectedType ← match expectedType? with
    | none              => mkFreshTypeMVar
    | some expectedType => pure expectedType
  mkSyntheticSorry expectedType
```

```
private def exceptionToSorry (ex : Exception) (expectedType? : Option Expr) : TermElabM Expr := do
  let syntheticSorry ← mkSyntheticSorryFor expectedType?
  logException ex
  pure syntheticSorry

/-- If `mayPostpone == true`, throw `Expection.postpone`. -/
def tryPostpone : TermElabM Unit := do
  if (← read).mayPostpone then
    throwPostpone

/-- If `mayPostpone == true` and `e`'s head is a metavariable, throw `Exception.postpone`. -/
def tryPostponeIfMVar (e : Expr) : TermElabM Unit := do
  if e.getAppFn.isMVar then
    let e ← instantiateMVars e
    if e.getAppFn.isMVar then
      tryPostpone

def tryPostponeIfNoneOrMVar (e? : Option Expr) : TermElabM Unit :=
  match e? with
  | some e => tryPostponeIfMVar e
  | none   => tryPostpone

def tryPostponeIfHasMVars (expectedType? : Option Expr) (msg : String) : TermElabM Expr := do
  tryPostponeIfNoneOrMVar expectedType?
  let some expectedType ← pure expectedType? |
    throwError "{msg}, expected type must be known"
  let expectedType ← instantiateMVars expectedType
  if expectedType.hasExprMVar then
    tryPostpone
    throwError "{msg}, expected type contains metavariables{indentExpr expectedType}"
  pure expectedType

private def saveContext : TermElabM SavedContext :=
  return {
    macroStack := (← read).macroStack
    declName?  := (← read).declName?
    options    := (← getOptions)
    openDecls  := (← getOpenDecls)
    errToSorry := (← read).errToSorry
  }

def withSavedContext (savedCtx : SavedContext) (x : TermElabM α) : TermElabM α := do
  withReader (fun ctx => { ctx with declName? := savedCtx.declName?, macroStack := savedCtx.macroStack, errToSorry := savedCtx.errToSor
    withTheReader Core.Context (fun ctx => { ctx with options := savedCtx.options, openDecls := savedCtx.openDecls })
      x
```

```
private def postponeElabTerm (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
  trace[Elab.postpone] "{stx} : {expectedType?}"
  let mvar ← mkFreshExprMVar expectedType? MetavarKind.syntheticOpaque
  let ctx ← read
  registerSyntheticMVar stx mvar.mvarId! (SyntheticMVarKind.postponed (← saveContext))
  pure mvar

/-
  Helper function for `elabTerm` is tries the registered elaboration functions for `stxNode` kind until it finds one that supports the
  an error is found. -/
private def elabUsingElabFnsAux (s : SavedState) (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone : Bool)
    : List TermElab → TermElabM Expr
  | []                => do throwError "unexpected syntax{indentD stx}"
  | (elabFn::elabFns) => do
    try
      elabFn stx expectedType?
    catch ex => match ex with
      | Exception.error _ _ =>
        if (← read).errToSorry then
          exceptionToSorry ex expectedType?
        else
          throw ex
      | Exception.internal id _ =>
        if (← read).errToSorry && id == abortTermExceptionId then
          exceptionToSorry ex expectedType?
        else if id == unsupportedSyntaxExceptionId then
          s.restore
          elabUsingElabFnsAux s stx expectedType? catchExPostpone elabFns
        else if catchExPostpone && id == postponeExceptionId then
          /- If `elab` threw `Exception.postpone`, we reset any state modifications.
             For example, we want to make sure pending synthetic metavariables created by `elab` before
             it threw `Exception.postpone` are discarded.
             Note that we are also discarding the messages created by `elab`.

             For example, consider the expression.
             `((f.x a1).x a2).x a3`
             Now, suppose the elaboration of `f.x a1` produces an `Exception.postpone`.
             Then, a new metavariable `?m` is created. Then, `?m.x a2` also throws `Exception.postpone`
             because the type of `?m` is not yet known. Then another, metavariable `?n` is created, and
            finally `?n.x a3` also throws `Exception.postpone`. If we did not restore the state, we would
            keep "dead" metavariables `?m` and `?n` on the pending synthetic metavariable list. This is
            wasteful because when we resume the elaboration of `((f.x a1).x a2).x a3`, we start it from scratch
            and new metavariables are created for the nested functions. -/
            s.restore
            postponeElabTerm stx expectedType?
          else
            throw ex
```

```
private def elabUsingElabFns (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone : Bool) : TermElabM Expr := do
  let s ← saveState
  let table := termElabAttribute.ext.getState (← getEnv) |>.table
  let k := stx.getKind
  match table.find? k with
  | some elabFns => elabUsingElabFnsAux s stx expectedType? catchExPostpone elabFns
  | none         => throwError "elaboration function for '{k}' has not been implemented{indentD stx}"

instance : MonadMacroAdapter TermElabM where
  getCurrMacroScope := getCurrMacroScope
  getNextMacroScope := return (← getThe Core.State).nextMacroScope
  setNextMacroScope next := modifyThe Core.State fun s => { s with nextMacroScope := next }

private def isExplicit (stx : Syntax) : Bool :=
  match stx with
  | `(@$f) => true
  | _      => false

private def isExplicitApp (stx : Syntax) : Bool :=
  stx.getKind == ``Lean.Parser.Term.app && isExplicit stx[0]

/--
  Return true if `stx` if a lambda abstraction containing a `{}` or `[]` binder annotation.
  Example: `fun {α} (a : α) => a` -/
private def isLambdaWithImplicit (stx : Syntax) : Bool :=
  match stx with
  | `(fun $binders* => $body) => binders.any fun b => b.isOfKind ``Lean.Parser.Term.implicitBinder || b.isOfKind `Lean.Parser.Term.inst
  | _                         => false

private partial def dropTermParens : Syntax → Syntax := fun stx =>
  match stx with
  | `(($stx)) => dropTermParens stx
  | _         => stx

private def isHole (stx : Syntax) : Bool :=
  match stx with
  | `(_)          => true
  | `(? _)        => true
  | `(? $x:ident) => true
  | _             => false

private def isTacticBlock (stx : Syntax) : Bool :=
  match stx with
  | `(by $x:tacticSeq) => true
  | _ => false
```

```
private def isNoImplicitLambda (stx : Syntax) : Bool :=
  match stx with
  | `(noImplicitLambda% $x:term) => true
  | _ => false

def mkNoImplicitLambdaAnnotation (type : Expr) : Expr :=
  mkAnnotation `noImplicitLambda type

def hasNoImplicitLambdaAnnotation (type : Expr) : Bool :=
  annotation? `noImplicitLambda type |>.isSome

/-- Block usage of implicit lambdas if `stx` is `@f` or `@f arg1 ...` or `fun` with an implicit binder annotation. -/
def blockImplicitLambda (stx : Syntax) : Bool :=
  let stx := dropTermParens stx
  -- TODO: make it extensible
  isExplicit stx || isExplicitApp stx || isLambdaWithImplicit stx || isHole stx || isTacticBlock stx || isNoImplicitLambda stx

/--
  Return normalized expected type if it is of the form `{a : α} → β` or `[a : α] → β` and
  `blockImplicitLambda stx` is not true, else return `none`. -/
private def useImplicitLambda? (stx : Syntax) (expectedType? : Option Expr) : TermElabM (Option Expr) :=
  if blockImplicitLambda stx then
    pure none
  else match expectedType? with
    | some expectedType => do
      if hasNoImplicitLambdaAnnotation expectedType then
        pure none
      else
        let expectedType ← whnfForall expectedType
        match expectedType with
        | Expr.forallE _ _ _ c => if c.binderInfo.isExplicit then pure none else pure $ some expectedType
        | _                    => pure none
    | _          => pure none

private def elabImplicitLambdaAux (stx : Syntax) (catchExPostpone : Bool) (expectedType : Expr) (fvars : Array Expr) : TermElabM Expr :=
  let body ← elabUsingElabFns stx expectedType catchExPostpone
  let body ← ensureHasType expectedType body
  let r ← mkLambdaFVars fvars body
  trace[Elab.implicitForall] r
  pure r

private partial def elabImplicitLambda (stx : Syntax) (catchExPostpone : Bool) : Expr → Array Expr → TermElabM Expr
  | type@(Expr.forallE n d b c), fvars =>
    if c.binderInfo.isExplicit then
      elabImplicitLambdaAux stx catchExPostpone type fvars
    else withFreshMacroScope do
      let n ← MonadQuotation.addMacroScope n
```

```
      withLocalDecl n c.binderInfo d fun fvar => do
        let type ← whnfForall (b.instantiate1 fvar)
        elabImplicitLambda stx catchExPostpone type (fvars.push fvar)
  | type, fvars =>
    elabImplicitLambdaAux stx catchExPostpone type fvars

/- Main loop for `elabTerm` -/
private partial def elabTermAux (expectedType? : Option Expr) (catchExPostpone : Bool) (implicitLambda : Bool) : Syntax → TermElabM Exp
  | Syntax.missing => mkSyntheticSorryFor expectedType?
  | stx => withFreshMacroScope <| withIncRecDepth do
    trace[Elab.step] "expected type: {expectedType?}, term\n{stx}"
    checkMaxHeartbeats "elaborator"
    withNestedTraces do
    let env ← getEnv
    let stxNew? ← catchInternalId unsupportedSyntaxExceptionId
      (do let newStx ← adaptMacro (getMacros env) stx; pure (some newStx))
      (fun _ => pure none)
    match stxNew? with
    | some stxNew => withMacroExpansion stx stxNew $ elabTermAux expectedType? catchExPostpone implicitLambda stxNew
    | _ =>
      let implicit? ← if implicitLambda && (← read).implicitLambda then useImplicitLambda? stx expectedType? else pure none
      match implicit? with
      | some expectedType => elabImplicitLambda stx catchExPostpone expectedType #[]
      | none              => elabUsingElabFns stx expectedType? catchExPostpone

def addTermInfo (stx : Syntax) (e : Expr) : TermElabM Unit := do
  if (← getInfoState).enabled then
    pushInfoLeaf <| Info.ofTermInfo { lctx := (← getLCtx), expr := e, stx := stx }

def getSyntheticMVarDecl? (mvarId : MVarId) : TermElabM (Option SyntheticMVarDecl) :=
  return (← get).syntheticMVars.find? fun d => d.mvarId == mvarId

def mkTermInfo (stx : Syntax) (e : Expr) : TermElabM (Sum Info MVarId) := do
  let isHole? : TermElabM (Option MVarId) := do
    match e with
    | Expr.mvar mvarId _ =>
      match (← getSyntheticMVarDecl? mvarId) with
      | some { kind := SyntheticMVarKind.tactic .., .. }    => return mvarId
      | some { kind := SyntheticMVarKind.postponed .., .. } => return mvarId
      | _                                                   => return none
    | _ => pure none
  match (← isHole?) with
  | none        => return Sum.inl <| Info.ofTermInfo { lctx := (← getLCtx), expr := e, stx := stx }
  | some mvarId => return Sum.inr mvarId

/-- Store in the `InfoTree` that `e` is a "dot"-completion target. -/
def addDotCompletionInfo (stx : Syntax) (e : Expr) (expectedType? : Option Expr) (field? : Option Syntax := none) : TermElabM Unit := d
```

```
  addCompletionInfo <| CompletionInfo.dot { expr := e, stx := stx, lctx := (← getLCtx) } (field? := field?) (expectedType? := expectedT

/--
  Main function for elaborating terms.
  It extracts the elaboration methods from the environment using the node kind.
  Recall that the environment has a mapping from `SyntaxNodeKind` to `TermElab` methods.
  It creates a fresh macro scope for executing the elaboration method.
  All unlogged trace messages produced by the elaboration method are logged using
  the position information at `stx`. If the elaboration method throws an `Exception.error` and `errToSorry == true`,
  the error is logged and a synthetic sorry expression is returned.
  If the elaboration throws `Exception.postpone` and `catchExPostpone == true`,
  a new synthetic metavariable of kind `SyntheticMVarKind.postponed` is created, registered,
  and returned.
  The option `catchExPostpone == false` is used to implement `resumeElabTerm`
  to prevent the creation of another synthetic metavariable when resuming the elaboration.

  If `implicitLambda == true`, then disable implicit lambdas feature for the given syntax, but not for its subterms.
  We use this flag to implement, for example, the `@` modifier. If `Context.implicitLambda == false`, then this parameter has no effect
  -/
def elabTerm (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone := true) (implicitLambda := true) : TermElabM Expr :=
  withInfoContext' (withRef stx <| elabTermAux expectedType? catchExPostpone implicitLambda stx) (mkTermInfo stx)

def elabTermEnsuringType (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone := true) (implicitLambda := true) (errorMsgHeade
  let e ← elabTerm stx expectedType? catchExPostpone implicitLambda
  withRef stx <| ensureHasType expectedType? e errorMsgHeader?

/-- Adapt a syntax transformation to a regular, term-producing elaborator. -/
def adaptExpander (exp : Syntax → TermElabM Syntax) : TermElab := fun stx expectedType? => do
  let stx' ← exp stx
  withMacroExpansion stx stx' $ elabTerm stx' expectedType?

def mkInstMVar (type : Expr) : TermElabM Expr := do
  let mvar ← mkFreshExprMVar type MetavarKind.synthetic
  let mvarId := mvar.mvarId!
  unless (← synthesizeInstMVarCore mvarId) do
    registerSyntheticMVarWithCurrRef mvarId SyntheticMVarKind.typeClass
  pure mvar

/-
  Relevant definitions:
  ```
  class CoeSort (α : Sort u) (β : outParam (Sort v))
  abbrev coeSort {α : Sort u} {β : Sort v} (a : α) [CoeSort α β] : β
  ```
  -/
private def tryCoeSort (α : Expr) (a : Expr) : TermElabM Expr := do
  let β ← mkFreshTypeMVar
```

```
  let u ← getLevel α
  let v ← getLevel β
  let coeSortInstType := mkAppN (Lean.mkConst ``CoeSort [u, v]) #[α, β]
  let mvar ← mkFreshExprMVar coeSortInstType MetavarKind.synthetic
  let mvarId := mvar.mvarId!
  try
    withoutMacroStackAtErr do
      if (← synthesizeCoeInstMVarCore mvarId) then
        expandCoe <| mkAppN (Lean.mkConst ``coeSort [u, v]) #[α, β, a, mvar]
      else
        throwError "type expected"
  catch
    | Exception.error _ msg => throwError "type expected\n{msg}"
    | _                     => throwError "type expected"

/--
  Make sure `e` is a type by inferring its type and making sure it is a `Expr.sort`
  or is unifiable with `Expr.sort`, or can be coerced into one. -/
def ensureType (e : Expr) : TermElabM Expr := do
  if (← isType e) then
    pure e
  else
    let eType ← inferType e
    let u ← mkFreshLevelMVar
    if (← isDefEq eType (mkSort u)) then
      pure e
    else
      tryCoeSort eType e

/-- Elaborate `stx` and ensure result is a type. -/
def elabType (stx : Syntax) : TermElabM Expr := do
  let u ← mkFreshLevelMVar
  let type ← elabTerm stx (mkSort u)
  withRef stx $ ensureType type

/--
  Enable auto-bound implicits, and execute `k` while catching auto bound implicit exceptions. When an exception is caught,
  a new local declaration is created, registered, and `k` is tried to be executed again. -/
partial def withAutoBoundImplicit (k : TermElabM α) : TermElabM α := do
  let flag := autoBoundImplicitLocal.get (← getOptions)
  if flag then
    withReader (fun ctx => { ctx with autoBoundImplicit := flag, autoBoundImplicits := {} }) do
      let rec loop (s : SavedState) : TermElabM α := do
        try
          k
        catch
          | ex => match isAutoBoundImplicitLocalException? ex with
```

```
          | some n =>
            -- Restore state, declare `n`, and try again
            s.restore
            withLocalDecl n BinderInfo.implicit (← mkFreshTypeMVar) fun x =>
              withReader (fun ctx => { ctx with autoBoundImplicits := ctx.autoBoundImplicits.push x } ) do
                loop (← saveState)
          | none   => throw ex
      loop (← saveState)
  else
    k

def withoutAutoBoundImplicit (k : TermElabM α) : TermElabM α := do
  withReader (fun ctx => { ctx with autoBoundImplicit := false, autoBoundImplicits := {} }) k

/--
  Return `autoBoundImplicits ++ xs.
  This methoid throws an error if a variable in `autoBoundImplicits` depends on some `x` in `xs` -/
def addAutoBoundImplicits (xs : Array Expr) : TermElabM (Array Expr) := do
  let autoBoundImplicits := (← read).autoBoundImplicits
  for auto in autoBoundImplicits do
    let localDecl ← getLocalDecl auto.fvarId!
    for x in xs do
      if (← getMCtx).localDeclDependsOn localDecl x.fvarId! then
        throwError "invalid auto implicit argument '{auto}', it depends on explicitly provided argument '{x}'"
  return autoBoundImplicits.toArray ++ xs

def mkAuxName (suffix : Name) : TermElabM Name := do
  match (← read).declName? with
  | none          => throwError "auxiliary declaration cannot be created when declaration name is not available"
  | some declName => Lean.mkAuxName (declName ++ suffix) 1

builtin_initialize registerTraceClass `Elab.letrec

/- Return true if mvarId is an auxiliary metavariable created for compiling `let rec` or it
   is delayed assigned to one. -/
def isLetRecAuxMVar (mvarId : MVarId) : TermElabM Bool := do
  trace[Elab.letrec] "mvarId: {mkMVar mvarId} letrecMVars: {(← get).letRecsToLift.map (mkMVar $ ·.mvarId)}"
  let mvarId := (← getMCtx).getDelayedRoot mvarId
  trace[Elab.letrec] "mvarId root: {mkMVar mvarId}"
  return (← get).letRecsToLift.any (·.mvarId == mvarId)

/- =======================================
       Builtin elaboration functions
   ======================================= -/

@[builtinTermElab «prop»] def elabProp : TermElab := fun _ _ =>
  return mkSort levelZero
```

```
private def elabOptLevel (stx : Syntax) : TermElabM Level :=
  if stx.isNone then
    pure levelZero
  else
    elabLevel stx[0]

@[builtinTermElab «sort»] def elabSort : TermElab := fun stx _ =>
  return mkSort (← elabOptLevel stx[1])

@[builtinTermElab «type»] def elabTypeStx : TermElab := fun stx _ =>
  return mkSort (mkLevelSucc (← elabOptLevel stx[1]))

/-
 the method `resolveName` adds a completion point for it using the given
    expected type. Thus, we propagate the expected type if `stx[0]` is an identifier.
    It doesn't "hurt" if the identifier can be resolved because the expected type is not used in this case.
    Recall that if the name resolution fails a synthetic sorry is returned.-/

@[builtinTermElab «pipeCompletion»] def elabPipeCompletion : TermElab := fun stx expectedType? => do
  let e ← elabTerm stx[0] none
  unless e.isSorry do
    addDotCompletionInfo stx e expectedType?
  throwErrorAt stx[1] "invalid field notation, identifier or numeral expected"

@[builtinTermElab «completion»] def elabCompletion : TermElab := fun stx expectedType? => do
  /- `ident.` is ambiguous in Lean, we may try to be completing a declaration name or access a "field". -/
  if stx[0].isIdent then
    /- If we can elaborate the identifier successfully, we assume it a dot-completion. Otherwise, we treat it as
       identifier completion with a dangling `.`.
       Recall that the server falls back to identifier completion when dot-completion fails. -/
    let s ← saveState
    try
      let e ← elabTerm stx[0] none
      addDotCompletionInfo stx e expectedType?
    catch _ =>
      s.restore
      addCompletionInfo <| CompletionInfo.id stx stx[0].getId (danglingDot := true) (← getLCtx) expectedType?
    throwErrorAt stx[1] "invalid field notation, identifier or numeral expected"
  else
    elabPipeCompletion stx expectedType?

@[builtinTermElab «hole»] def elabHole : TermElab := fun stx expectedType? => do
  let mvar ← mkFreshExprMVar expectedType?
  registerMVarErrorHoleInfo mvar.mvarId! stx
  pure mvar
```

```
@[builtinTermElab «syntheticHole»] def elabSyntheticHole : TermElab := fun stx expectedType? => do
  let arg  := stx[1]
  let userName := if arg.isIdent then arg.getId else Name.anonymous
  let mkNewHole : Unit → TermElabM Expr := fun _ => do
    let mvar ← mkFreshExprMVar expectedType? MetavarKind.syntheticOpaque userName
    registerMVarErrorHoleInfo mvar.mvarId! stx
    pure mvar
  if userName.isAnonymous then
    mkNewHole ()
  else
    let mctx ← getMCtx
    match mctx.findUserName? userName with
    | none => mkNewHole ()
    | some mvarId =>
      let mvar := mkMVar mvarId
      let mvarDecl ← getMVarDecl mvarId
      let lctx ← getLCtx
      if mvarDecl.lctx.isSubPrefixOf lctx then
        pure mvar
      else match mctx.getExprAssignment? mvarId with
      | some val =>
        let val ← instantiateMVars val
        if mctx.isWellFormed lctx val then
          pure val
        else
          withLCtx mvarDecl.lctx mvarDecl.localInstances do
            throwError "synthetic hole has already been defined and assigned to value incompatible with the current context{indentExpr 
      | none =>
        if mctx.isDelayedAssigned mvarId then
          -- We can try to improve this case if needed.
          throwError "synthetic hole has already beend defined and delayed assigned with an incompatible local context"
        else if lctx.isSubPrefixOf mvarDecl.lctx then
          let mvarNew ← mkNewHole ()
          modifyMCtx fun mctx => mctx.assignExpr mvarId mvarNew
          pure mvarNew
        else
          throwError "synthetic hole has already been defined with an incompatible local context"

private def mkTacticMVar (type : Expr) (tacticCode : Syntax) : TermElabM Expr := do
  let mvar ← mkFreshExprMVar type MetavarKind.syntheticOpaque
  let mvarId := mvar.mvarId!
  let ref ← getRef
  let declName? ← getDeclName?
  registerSyntheticMVar ref mvarId <| SyntheticMVarKind.tactic tacticCode (← saveContext)
  return mvar

@[builtinTermElab byTactic] def elabByTactic : TermElab := fun stx expectedType? =>
```

```
  match expectedType? with
  | some expectedType => mkTacticMVar expectedType stx
  | none => throwError ("invalid 'by' tactic, expected type has not been provided")

@[builtinTermElab noImplicitLambda] def elabNoImplicitLambda : TermElab := fun stx expectedType? =>
  elabTerm stx[1] (mkNoImplicitLambdaAnnotation <$> expectedType?)

def resolveLocalName (n : Name) : TermElabM (Option (Expr × List String)) := do
  let lctx ← getLCtx
  let view := extractMacroScopes n
  let rec loop (n : Name) (projs : List String) :=
    match lctx.findFromUserName? { view with name := n }.review with
    | some decl => some (decl.toExpr, projs)
    | none      => match n with
      | Name.str pre s _ => loop pre (s::projs)
      | _                => none
  return loop view.name []

/- Return true iff `stx` is a `Syntax.ident`, and it is a local variable. -/
def isLocalIdent? (stx : Syntax) : TermElabM (Option Expr) :=
  match stx with
  | Syntax.ident _ _ val _ => do
    let r? ← resolveLocalName val
    match r? with
    | some (fvar, []) => pure (some fvar)
    | _               => pure none
  | _ => pure none

/--
  Create an `Expr.const` using the given name and explicit levels.
  Remark: fresh universe metavariables are created if the constant has more universe
  parameters than `explicitLevels`. -/
def mkConst (constName : Name) (explicitLevels : List Level := []) : TermElabM Expr := do
  let cinfo ← getConstInfo constName
  if explicitLevels.length > cinfo.levelParams.length then
    throwError "too many explicit universe levels"
  else
    let numMissingLevels := cinfo.levelParams.length - explicitLevels.length
    let us ← mkFreshLevelMVars numMissingLevels
    pure $ Lean.mkConst constName (explicitLevels ++ us)

private def mkConsts (candidates : List (Name × List String)) (explicitLevels : List Level) : TermElabM (List (Expr × List String)) :=
  candidates.foldlM (init := []) fun result (constName, projs) => do
    -- TODO: better suppor for `mkConst` failure. We may want to cache the failures, and report them if all candidates fail.
   let const ← mkConst constName explicitLevels
   return (const, projs) :: result

```

```
def resolveName (stx : Syntax) (n : Name) (preresolved : List (Name × List String)) (explicitLevels : List Level) (expectedType? : Opti
  try
    if let some (e, projs) ← resolveLocalName n then
      unless explicitLevels.isEmpty do
        throwError "invalid use of explicit universe parameters, '{e}' is a local"
      return [(e, projs)]
    -- check for section variable capture by a quotation
    let ctx ← read
    if let some (e, projs) := preresolved.findSome? fun (n, projs) => ctx.sectionFVars.find? n |>.map (·, projs) then
      return [(e, projs)]  -- section variables should shadow global decls
    if preresolved.isEmpty then
      process (← resolveGlobalName n)
    else
      process preresolved
  catch ex =>
    if preresolved.isEmpty && explicitLevels.isEmpty then
      addCompletionInfo <| CompletionInfo.id stx stx.getId (danglingDot := false) (← getLCtx) expectedType?
    throw ex
where process (candidates : List (Name × List String)) : TermElabM (List (Expr × List String)) := do
  if candidates.isEmpty then
    if (← read).autoBoundImplicit && isValidAutoBoundImplicitName n then
      throwAutoBoundImplicitLocal n
    else
      throwError "unknown identifier '{Lean.mkConst n}'"
  if preresolved.isEmpty && explicitLevels.isEmpty then
    addCompletionInfo <| CompletionInfo.id stx stx.getId (danglingDot := false) (← getLCtx) expectedType?
  mkConsts candidates explicitLevels

/--
  Similar to `resolveName`, but creates identifiers for the main part and each projection with position information derived from `ident
  Example: Assume resolveName `v.head.bla.boo` produces `(v.head, ["bla", "boo"])`, then this method produces
  `(v.head, id, [f₁, f₂])` where `id` is an identifier for `v.head`, and `f₁` and `f₂` are identifiers for fields `"bla"` and `"boo"`.
def resolveName' (ident : Syntax) (explicitLevels : List Level) (expectedType? : Option Expr := none) : TermElabM (List (Expr × Syntax 
  match ident with
  | Syntax.ident info rawStr n preresolved =>
    let r ← resolveName ident n preresolved explicitLevels expectedType?
    r.mapM fun (c, fields) => do
      let (cSstr, fields) := fields.foldr (init := (rawStr, [])) fun field (restSstr, fs) =>
        let fieldSstr := restSstr.takeRightWhile (· ≠ '.')
        ({ restSstr with stopPos := restSstr.stopPos - (fieldSstr.bsize + 1) }, (field, fieldSstr) :: fs)
      let mkIdentFromPos pos rawVal val :=
        let info := match info with
        | SourceInfo.original .. => SourceInfo.original "".toSubstring pos "".toSubstring
        | _                      => SourceInfo.synthetic pos (pos + rawVal.bsize)
        Syntax.ident info rawVal val []
      let id := match c with
        | Expr.const id _ _ => id
```

```
        | Expr.fvar id _    => id
        | _                 => unreachable!
      let id := mkIdentFromPos (ident.getPos?.getD 0) cSstr id
      match info.getPos? with
      | none =>
        return (c, id, fields.map fun (field, _) => mkIdentFrom ident (Name.mkSimple field))
      | some pos =>
        let mut pos := pos + cSstr.bsize + 1
        let mut newFields := #[]
        for (field, fieldSstr) in fields do
          newFields := newFields.push <| mkIdentFromPos pos fieldSstr (Name.mkSimple field)
          pos := pos + fieldSstr.bsize + 1
        return (c, id, newFields.toList)
  | _ => throwError "identifier expected"

def resolveId? (stx : Syntax) (kind := "term") : TermElabM (Option Expr) :=
  match stx with
  | Syntax.ident _ _ val preresolved => do
    let rs ← try resolveName stx val preresolved [] catch _ => pure []
    let rs := rs.filter fun ⟨f, projs⟩ => projs.isEmpty
    let fs := rs.map fun (f, _) => f
    match fs with
    | []  => pure none
    | [f] => pure (some f)
    | _   =>   throwError "ambiguous {kind}, use fully qualified name, possible interpretations {fs}"
  | _ => throwError "identifier expected"

@[builtinTermElab cdot] def elabBadCDot : TermElab := fun stx _ =>
  throwError "invalid occurrence of `·` notation, it must be surrounded by parentheses (e.g. `(· + 1)`)"

@[builtinTermElab strLit] def elabStrLit : TermElab := fun stx _ => do
  match stx.isStrLit? with
  | some val => pure $ mkStrLit val
  | none     => throwIllFormedSyntax

private def mkFreshTypeMVarFor (expectedType? : Option Expr) : TermElabM Expr := do
  let typeMVar ← mkFreshTypeMVar MetavarKind.synthetic
  match expectedType? with
  | some expectedType => discard <| isDefEq expectedType typeMVar
  | _                 => pure ()
  return typeMVar

@[builtinTermElab numLit] def elabNumLit : TermElab := fun stx expectedType? => do
  let val ← match stx.isNatLit? with
    | some val => pure val
    | none     => throwIllFormedSyntax
  let typeMVar ← mkFreshTypeMVarFor expectedType?
```

```
  let u ← getDecLevel typeMVar
  let mvar ← mkInstMVar (mkApp2 (Lean.mkConst ``OfNat [u]) typeMVar (mkNatLit val))
  let r := mkApp3 (Lean.mkConst ``OfNat.ofNat [u]) typeMVar (mkNatLit val) mvar
  registerMVarErrorImplicitArgInfo mvar.mvarId! stx r
  return r

@[builtinTermElab rawNatLit] def elabRawNatLit : TermElab :=  fun stx expectedType? => do
  match stx[1].isNatLit? with
  | some val => return mkNatLit val
  | none     => throwIllFormedSyntax

@[builtinTermElab scientificLit]
def elabScientificLit : TermElab := fun stx expectedType? => do
  match stx.isScientificLit? with
  | none        => throwIllFormedSyntax
  | some (m, sign, e) =>
    let typeMVar ← mkFreshTypeMVarFor expectedType?
    let u ← getDecLevel typeMVar
    let mvar ← mkInstMVar (mkApp (Lean.mkConst ``OfScientific [u]) typeMVar)
    return mkApp5 (Lean.mkConst ``OfScientific.ofScientific [u]) typeMVar mvar (mkNatLit m) (toExpr sign) (mkNatLit e)

@[builtinTermElab charLit] def elabCharLit : TermElab := fun stx _ => do
  match stx.isCharLit? with
  | some val => return mkApp (Lean.mkConst ``Char.ofNat) (mkNatLit val.toNat)
  | none     => throwIllFormedSyntax

@[builtinTermElab quotedName] def elabQuotedName : TermElab := fun stx _ =>
  match stx[0].isNameLit? with
  | some val => pure $ toExpr val
  | none     => throwIllFormedSyntax

@[builtinTermElab doubleQuotedName] def elabDoubleQuotedName : TermElab := fun stx _ => do
  match stx[1].isNameLit? with
  | some val => toExpr (← resolveGlobalConstNoOverloadWithInfo stx[1] val)
  | none     => throwIllFormedSyntax

@[builtinTermElab typeOf] def elabTypeOf : TermElab := fun stx _ => do
  inferType (← elabTerm stx[1] none)

@[builtinTermElab ensureTypeOf] def elabEnsureTypeOf : TermElab := fun stx expectedType? =>
  match stx[2].isStrLit? with
  | none     => throwIllFormedSyntax
  | some msg => do
    let refTerm ← elabTerm stx[1] none
    let refTermType ← inferType refTerm
    elabTermEnsuringType stx[3] refTermType (errorMsgHeader? := msg)
```

```
@[builtinTermElab ensureExpectedType] def elabEnsureExpectedType : TermElab := fun stx expectedType? =>
  match stx[1].isStrLit? with
  | none     => throwIllFormedSyntax
  | some msg => elabTermEnsuringType stx[2] expectedType? (errorMsgHeader? := msg)

@[builtinTermElab «open»] def elabOpen : TermElab := fun stx expectedType? => do
  try
    pushScope
    let openDecls ← elabOpenDecl stx[1]
    withTheReader Core.Context (fun ctx => { ctx with openDecls := openDecls }) do
      elabTerm stx[3] expectedType?
  finally
    popScope

@[builtinTermElab «set_option»] def elabSetOption : TermElab := fun stx expectedType? => do
  let options ← Elab.elabSetOption stx[1] stx[2]
  withTheReader Core.Context (fun ctx => { ctx with maxRecDepth := maxRecDepth.get options, options := options }) do
    elabTerm stx[4] expectedType?

private def mkSomeContext : Context := {
  fileName      := "<TermElabM>"
  fileMap       := arbitrary
}

@[inline] def TermElabM.run (x : TermElabM α) (ctx : Context := mkSomeContext) (s : State := {}) : MetaM (α × State) :=
  withConfig setElabConfig (x ctx |>.run s)

@[inline] def TermElabM.run' (x : TermElabM α) (ctx : Context := mkSomeContext) (s : State := {}) : MetaM α :=
  (·.1) <$> x.run ctx s

@[inline] def TermElabM.toIO (x : TermElabM α)
    (ctxCore : Core.Context) (sCore : Core.State)
    (ctxMeta : Meta.Context) (sMeta : Meta.State)
    (ctx : Context) (s : State) : IO (α × Core.State × Meta.State × State) := do
  let ((a, s), sCore, sMeta) ← (x.run ctx s).toIO ctxCore sCore ctxMeta sMeta
  pure (a, sCore, sMeta, s)

instance [MetaEval α] : MetaEval (TermElabM α) where
  eval env opts x _ :=
    let x : TermElabM α := do
      try x finally
        let s ← get
        s.messages.forM fun msg => do IO.println (← msg.toString)
    MetaEval.eval env opts (hideUnit := true) $ x.run' mkSomeContext

unsafe def evalExpr (α) (typeName : Name) (value : Expr) : TermElabM α :=
  withoutModifyingEnv do
```

```
    let name ← mkFreshUserName `_tmp
    let type ← inferType value
    let type ← whnfD type
    unless type.isConstOf typeName do
      throwError "unexpected type at evalExpr{indentExpr type}"
    let decl := Declaration.defnDecl {
       name := name, levelParams := [], type := type,
       value := value, hints := ReducibilityHints.opaque,
       safety := DefinitionSafety.unsafe
    }
    ensureNoUnassignedMVars decl
    addAndCompile decl
    evalConst α name

private def throwStuckAtUniverseCnstr : TermElabM Unit := do
  let entries ← getPostponed
  let mut found : Std.HashSet (Level × Level) := {}
  let mut uniqueEntries := #[]
  for entry in entries do
    let mut lhs := entry.lhs
    let mut rhs := entry.rhs
    if Level.normLt rhs lhs then
      (lhs, rhs) := (rhs, lhs)
    unless found.contains (lhs, rhs) do
      found := found.insert (lhs, rhs)
      uniqueEntries := uniqueEntries.push entry
  for i in [1:uniqueEntries.size] do
    logErrorAt uniqueEntries[i].ref (← mkMessage uniqueEntries[i])
  throwErrorAt uniqueEntries[0].ref (← mkMessage uniqueEntries[0])
where
  /- Annotate any constant and sort in `e` that satisfies `p` with `pp.universes true` -/
  exposeRelevantUniverses (e : Expr) (p : Level → Bool) : Expr :=
    e.replace fun e =>
      match e with
      | Expr.const _ us _ => if us.any p then some (e.setPPUniverses true) else none
      | Expr.sort u _     => if p u then some (e.setPPUniverses true) else none
      | _                 => none

  mkMessage (entry : PostponedEntry) : TermElabM MessageData := do
    match entry.ctx? with
    | none =>
      return m!"stuck at solving universe constraints{indentD m!"{entry.lhs} =?= {entry.rhs}"}"
    | some ctx =>
      withLCtx ctx.lctx ctx.localInstances do
        let s   := entry.lhs.collectMVars entry.rhs.collectMVars
        /- `p u` is true if it contains a universe metavariable in `s` -/
        let p (u : Level) := u.any fun | Level.mvar m _ => s.contains m | _ => false
```

```
      let lhs := exposeRelevantUniverses (← instantiateMVars ctx.lhs) p
      let rhs := exposeRelevantUniverses (← instantiateMVars ctx.rhs) p
      addMessageContext m!"stuck at solving universe constraints{indentD m!"{entry.lhs} =?= {entry.rhs}"}\nwhile trying to unify{inde

@[specialize] def withoutPostponingUniverseConstraints (x : TermElabM α) : TermElabM α := do
  let postponed ← getResetPostponed
  try
    let a ← x
    unless (← processPostponed (mayPostpone := false)) do
      throwStuckAtUniverseCnstr
    setPostponed postponed
    return a
  catch ex =>
    setPostponed postponed
    throw ex

end Term

builtin_initialize
  registerTraceClass `Elab.postpone
  registerTraceClass `Elab.coe
  registerTraceClass `Elab.debug

export Term (TermElabM)

end Lean.Elab
::::::::::::::
Elab/Util.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.Trace
import Lean.Parser.Syntax
import Lean.Parser.Extension
import Lean.KeyedDeclsAttribute
import Lean.Elab.Exception

namespace Lean

def Syntax.prettyPrint (stx : Syntax) : Format :=
  match stx.unsetTrailing.reprint with -- TODO use syntax pretty printer
  | some str => format str.toFormat
  | none     => format stx

```

```
def MacroScopesView.format (view : MacroScopesView) (mainModule : Name) : Format :=
  fmt $
    if view.scopes.isEmpty then
      view.name
    else if view.mainModule == mainModule then
      view.scopes.foldl Name.mkNum (view.name ++ view.imported)
    else
      view.scopes.foldl Name.mkNum (view.name ++ view.imported ++ view.mainModule)

namespace Elab

def expandOptNamedPrio (stx : Syntax) : MacroM Nat :=
  if stx.isNone then
    return eval_prio default
  else match stx[0] with
    | `(Parser.Command.namedPrio| (priority := $prio)) => evalPrio prio
    | _ => Macro.throwUnsupported

def expandOptNamedName (stx : Syntax) : MacroM (Option Name) := do
  if stx.isNone then
    return none
  else match stx[0] with
    | `(Parser.Command.namedName| (name := $name)) => return name.getId
    | _ => Macro.throwUnsupported

structure MacroStackElem where
  before : Syntax
  after : Syntax

abbrev MacroStack := List MacroStackElem

/- If `ref` does not have position information, then try to use macroStack -/
def getBetterRef (ref : Syntax) (macroStack : MacroStack) : Syntax :=
  match ref.getPos? with
  | some _ => ref
  | none   =>
    match macroStack.find? (·.before.getPos? != none) with
    | some elem => elem.before
    | none      => ref

register_builtin_option pp.macroStack : Bool := {
  defValue := false
  group    := "pp"
  descr    := "dispaly macro expansion stack"
}

def addMacroStack {m} [Monad m] [MonadOptions m] (msgData : MessageData) (macroStack : MacroStack) : m MessageData := do
```

```
  if !pp.macroStack.get (← getOptions) then pure msgData else
  match macroStack with
  | []             => pure msgData
  | stack@(top::_) =>
    let msgData := msgData ++ Format.line ++ "with resulting expansion" ++ indentD top.after
    pure $ stack.foldl
      (fun (msgData : MessageData) (elem : MacroStackElem) =>
        msgData ++ Format.line ++ "while expanding" ++ indentD elem.before)
      msgData

def checkSyntaxNodeKind (k : Name) : AttrM Name := do
  if Parser.isValidSyntaxNodeKind (← getEnv) k then pure k
  else throwError "failed"

def checkSyntaxNodeKindAtNamespacesAux (k : Name) : Name → AttrM Name
  | n@(Name.str p _ _) => checkSyntaxNodeKind (n ++ k) <|> checkSyntaxNodeKindAtNamespacesAux k p
  | _ => throwError "failed"

def checkSyntaxNodeKindAtNamespaces (k : Name) : AttrM Name := do
  let ctx ← read
  checkSyntaxNodeKindAtNamespacesAux k ctx.currNamespace

def syntaxNodeKindOfAttrParam (defaultParserNamespace : Name) (stx : Syntax) : AttrM SyntaxNodeKind := do
  let k ← Attribute.Builtin.getId stx
  checkSyntaxNodeKind k
  <|>
  checkSyntaxNodeKindAtNamespaces k
  <|>
  checkSyntaxNodeKind (defaultParserNamespace ++ k)
  <|>
  throwError "invalid syntax node kind '{k}'"

private unsafe def evalSyntaxConstantUnsafe (env : Environment) (opts : Options) (constName : Name) : ExceptT String Id Syntax :=
  env.evalConstCheck Syntax opts `Lean.Syntax constName

@[implementedBy evalSyntaxConstantUnsafe]
constant evalSyntaxConstant (env : Environment) (opts : Options) (constName : Name) : ExceptT String Id Syntax := throw ""

unsafe def mkElabAttribute (γ) (attrDeclName attrBuiltinName attrName : Name) (parserNamespace : Name) (typeName : Name) (kind : String
    : IO (KeyedDeclsAttribute γ) :=
  KeyedDeclsAttribute.init {
    builtinName   := attrBuiltinName
    name          := attrName
    descr         := kind ++ " elaborator"
    valueTypeName := typeName
    evalKey       := fun _ stx => syntaxNodeKindOfAttrParam parserNamespace stx
  } attrDeclName
```

```
unsafe def mkMacroAttributeUnsafe : IO (KeyedDeclsAttribute Macro) :=
  mkElabAttribute Macro `Lean.Elab.macroAttribute `builtinMacro `macro Name.anonymous `Lean.Macro "macro"

@[implementedBy mkMacroAttributeUnsafe]
constant mkMacroAttribute : IO (KeyedDeclsAttribute Macro)

builtin_initialize macroAttribute : KeyedDeclsAttribute Macro ← mkMacroAttribute

private def expandMacroFns (stx : Syntax) : List Macro → MacroM Syntax
  | []    => throw Macro.Exception.unsupportedSyntax
  | m::ms => do
    try
      m stx
    catch
      | Macro.Exception.unsupportedSyntax => expandMacroFns stx ms
      | ex                                => throw ex

def getMacros (env : Environment) : Macro := fun stx =>
  let k := stx.getKind
  let table := (macroAttribute.ext.getState env).table
  match table.find? k with
  | some macroFns => expandMacroFns stx macroFns
  | none          => throw Macro.Exception.unsupportedSyntax

class MonadMacroAdapter (m : Type → Type) where
  getCurrMacroScope                  : m MacroScope
  getNextMacroScope                  : m MacroScope
  setNextMacroScope                  : MacroScope → m Unit

instance (m n) [MonadLift m n] [MonadMacroAdapter m] : MonadMacroAdapter n := {
  getCurrMacroScope := liftM (MonadMacroAdapter.getCurrMacroScope : m _),
  getNextMacroScope := liftM (MonadMacroAdapter.getNextMacroScope : m _),
  setNextMacroScope := fun s => liftM (MonadMacroAdapter.setNextMacroScope s : m _)
}

private def expandMacro? (env : Environment) (stx : Syntax) : MacroM (Option Syntax) := do
  try
    let newStx ← getMacros env stx
    pure (some newStx)
  catch
    | Macro.Exception.unsupportedSyntax => pure none
    | ex                                => throw ex

@[inline] def liftMacroM {α} {m : Type → Type} [Monad m] [MonadMacroAdapter m] [MonadEnv m] [MonadRecDepth m] [MonadError m] (x : MacroI
  let env  ← getEnv
  match x { macroEnv       := Macro.mkMacroEnv (expandMacro? env),
```

```
            ref            := ← getRef,
            currMacroScope := ← MonadMacroAdapter.getCurrMacroScope,
            mainModule     := env.mainModule,
            currRecDepth   := ← MonadRecDepth.getRecDepth,
            maxRecDepth    := ← MonadRecDepth.getMaxRecDepth } (← MonadMacroAdapter.getNextMacroScope) with
  | EStateM.Result.error Macro.Exception.unsupportedSyntax _ => throwUnsupportedSyntax
  | EStateM.Result.error (Macro.Exception.error ref msg) _   => throwErrorAt ref msg
  | EStateM.Result.ok a nextMacroScope                       => MonadMacroAdapter.setNextMacroScope nextMacroScope; pure a

@[inline] def adaptMacro {m : Type → Type} [Monad m] [MonadMacroAdapter m] [MonadEnv m] [MonadRecDepth m] [MonadError m] (x : Macro) (s
  liftMacroM (x stx)

partial def mkUnusedBaseName [Monad m] [MonadEnv m] [MonadResolveName m] (baseName : Name) : m Name := do
  let currNamespace ← getCurrNamespace
  let env ← getEnv
  if env.contains (currNamespace ++ baseName) then
    let rec loop (idx : Nat) :=
       let name := baseName.appendIndexAfter idx
       if env.contains (currNamespace ++ name) then
         loop (idx+1)
       else
         name
    return loop 1
  else
    return baseName

builtin_initialize
  registerTraceClass `Elab
  registerTraceClass `Elab.step

end Lean.Elab
::::::::::::::
Parser.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Basic
import Lean.Parser.Level
import Lean.Parser.Term
import Lean.Parser.Tactic
import Lean.Parser.Command
import Lean.Parser.Module
import Lean.Parser.Syntax
import Lean.Parser.Do
```

```
namespace Lean
namespace Parser

builtin_initialize
  registerAlias "ws" checkWsBefore
  registerAlias "noWs" checkNoWsBefore
  registerAlias "linebreak" checkLinebreakBefore
  registerAlias "num" numLit
  registerAlias "str" strLit
  registerAlias "char" charLit
  registerAlias "name" nameLit
  registerAlias "ident" ident
  registerAlias "colGt" checkColGt
  registerAlias "colGe" checkColGe
  registerAlias "lookahead" lookahead
  registerAlias "atomic" atomic
  registerAlias "many" many
  registerAlias "many1" many1
  registerAlias "notFollowedBy" (notFollowedBy · "element")
  registerAlias "optional" optional
  registerAlias "withPosition" withPosition
  registerAlias "interpolatedStr" interpolatedStr
  registerAlias "orelse" orelse
  registerAlias "andthen" andthen

end Parser

namespace PrettyPrinter
namespace Parenthesizer

-- Close the mutual recursion loop; see corresponding `[extern]` in the parenthesizer.
@[export lean_mk_antiquot_parenthesizer]
def mkAntiquot.parenthesizer (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Parenthesizer :=
  Parser.mkAntiquot.parenthesizer name kind anonymous

-- The parenthesizer auto-generated these instances correctly, but tagged them with the wrong kind, since the actual kind
-- (e.g. `ident`) is not equal to the parser name `Lean.Parser.Term.ident`.
@[builtinParenthesizer ident] def ident.parenthesizer : Parenthesizer := Parser.Term.ident.parenthesizer
@[builtinParenthesizer numLit] def numLit.parenthesizer : Parenthesizer := Parser.Term.num.parenthesizer
@[builtinParenthesizer scientificLit] def scientificLit.parenthesizer : Parenthesizer := Parser.Term.scientific.parenthesizer
@[builtinParenthesizer charLit] def charLit.parenthesizer : Parenthesizer := Parser.Term.char.parenthesizer
@[builtinParenthesizer strLit] def strLit.parenthesizer : Parenthesizer := Parser.Term.str.parenthesizer

open Lean.Parser

@[export lean_pretty_printer_parenthesizer_interpret_parser_descr]
```

```
unsafe def interpretParserDescr : ParserDescr → CoreM Parenthesizer
  | ParserDescr.const n                          => getConstAlias parenthesizerAliasesRef n
  | ParserDescr.unary n d                        => return (← getUnaryAlias parenthesizerAliasesRef n) (← interpretParserDescr d)
  | ParserDescr.binary n d₁ d₂                   => return (← getBinaryAlias parenthesizerAliasesRef n) (← interpretParserDescr d₁)
  | ParserDescr.node k prec d                    => return leadingNode.parenthesizer k prec (← interpretParserDescr d)
  | ParserDescr.nodeWithAntiquot _ k d           => return node.parenthesizer k (← interpretParserDescr d)
  | ParserDescr.sepBy p sep psep trail           => return sepBy.parenthesizer (← interpretParserDescr p) sep (← interpretParserDesc
  | ParserDescr.sepBy1 p sep psep trail          => return sepBy1.parenthesizer (← interpretParserDescr p) sep (← interpretParserDes
  | ParserDescr.trailingNode k prec lhsPrec d    => return trailingNode.parenthesizer k prec lhsPrec (← interpretParserDescr d)
  | ParserDescr.symbol tk                        => return symbol.parenthesizer tk
  | ParserDescr.nonReservedSymbol tk includeIdent   => return nonReservedSymbol.parenthesizer tk includeIdent
  | ParserDescr.parser constName                 => combinatorParenthesizerAttribute.runDeclFor constName
  | ParserDescr.cat catName prec                 => return categoryParser.parenthesizer catName prec

end Parenthesizer

namespace Formatter

@[export lean_mk_antiquot_formatter]
def mkAntiquot.formatter (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Formatter :=
  Parser.mkAntiquot.formatter name kind anonymous

@[builtinFormatter ident] def ident.formatter : Formatter := Parser.Term.ident.formatter
@[builtinFormatter numLit] def numLit.formatter : Formatter := Parser.Term.num.formatter
@[builtinFormatter scientificLit] def scientificLit.formatter : Formatter := Parser.Term.scientific.formatter
@[builtinFormatter charLit] def charLit.formatter : Formatter := Parser.Term.char.formatter
@[builtinFormatter strLit] def strLit.formatter : Formatter := Parser.Term.str.formatter

open Lean.Parser

@[export lean_pretty_printer_formatter_interpret_parser_descr]
unsafe def interpretParserDescr : ParserDescr → CoreM Formatter
  | ParserDescr.const n                          => getConstAlias formatterAliasesRef n
  | ParserDescr.unary n d                        => return (← getUnaryAlias formatterAliasesRef n) (← interpretParserDescr d)
  | ParserDescr.binary n d₁ d₂                   => return (← getBinaryAlias formatterAliasesRef n) (← interpretParserDescr d₁) (← i
  | ParserDescr.node k prec d                    => return node.formatter k (← interpretParserDescr d)
  | ParserDescr.nodeWithAntiquot _ k d           => return node.formatter k (← interpretParserDescr d)
  | ParserDescr.sepBy p sep psep trail           => return sepBy.formatter (← interpretParserDescr p) sep (← interpretParserDescr ps
  | ParserDescr.sepBy1 p sep psep trail          => return sepBy1.formatter (← interpretParserDescr p) sep (← interpretParserDescr p
  | ParserDescr.trailingNode k prec lhsPrec d    => return trailingNode.formatter k prec lhsPrec (← interpretParserDescr d)
  | ParserDescr.symbol tk                        => return symbol.formatter tk
  | ParserDescr.nonReservedSymbol tk includeIdent   => return nonReservedSymbol.formatter tk
  | ParserDescr.parser constName                 => combinatorFormatterAttribute.runDeclFor constName
  | ParserDescr.cat catName prec                 => return categoryParser.formatter catName

end Formatter
end PrettyPrinter
```

```
end Lean
::::::::::::::
Parser/Attr.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Basic
import Lean.Parser.Extra

namespace Lean.Parser

builtin_initialize
  registerBuiltinParserAttribute `builtinPrioParser `prio LeadingIdentBehavior.both
  registerBuiltinDynamicParserAttribute `prioParser `prio

builtin_initialize
  registerBuiltinParserAttribute `builtinAttrParser `attr LeadingIdentBehavior.symbol
  registerBuiltinDynamicParserAttribute `attrParser `attr

@[inline] def priorityParser (rbp : Nat := 0) : Parser :=
  categoryParser `prio rbp

@[inline] def attrParser (rbp : Nat := 0) : Parser :=
  categoryParser `attr rbp

attribute [runBuiltinParserAttributeHooks]
  priorityParser attrParser

namespace Priority
@[builtinPrioParser] def numPrio  := checkPrec maxPrec >> numLit
attribute [runBuiltinParserAttributeHooks] numPrio
end Priority

namespace Attr

@[builtinAttrParser] def simple     := leading_parser ident >> optional (priorityParser <|> ident)
/- Remark, We can't use `simple` for `class`, `instance`, `export`, and `macro` because they are  keywords. -/
@[builtinAttrParser] def «macro»    := leading_parser "macro " >> ident
@[builtinAttrParser] def «export»   := leading_parser "export " >> ident

/- We don't use `simple` for recursor because the argument is not a priority-/
@[builtinAttrParser] def recursor        := leading_parser nonReservedSymbol "recursor " >> numLit
@[builtinAttrParser] def «class»         := leading_parser "class"
@[builtinAttrParser] def «instance»      := leading_parser "instance" >> optional priorityParser
```

```
@[builtinAttrParser] def defaultInstance := leading_parser nonReservedSymbol "defaultInstance " >> optional priorityParser

def externEntry := leading_parser optional ident >> optional (nonReservedSymbol "inline ") >> strLit
@[builtinAttrParser] def extern     := leading_parser nonReservedSymbol "extern " >> optional numLit >> many externEntry

end Attr

end Lean.Parser
::::::::::::::
Parser/Basic.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/

/-!
# Basic Lean parser infrastructure

The Lean parser was developed with the following primary goals in mind:

* flexibility: Lean's grammar is complex and includes indentation and other whitespace sensitivity. It should be
  possible to introduce such custom "tweaks" locally without having to adjust the fundamental parsing approach.
* extensibility: Lean's grammar can be extended on the fly within a Lean file, and with Lean 4 we want to extend this
  to cover embedding domain-specific languages that may look nothing like Lean, down to using a separate set of tokens.
* losslessness: The parser should produce a concrete syntax tree that preserves all whitespace and other "sub-token"
  information for the use in tooling.
* performance: The overhead of the parser building blocks, and the overall parser performance on average-complexity
  input, should be comparable with that of the previous parser hand-written in C++. No fancy optimizations should be
  necessary for this.

Given these constraints, we decided to implement a combinatoric, non-monadic, lexer-less, memoizing recursive-descent
parser. Using combinators instead of some more formal and introspectible grammar representation ensures ultimate
flexibility as well as efficient extensibility: there is (almost) no pre-processing necessary when extending the grammar
with a new parser. However, because the all results the combinators produce are of the homogeneous `Syntax` type, the
basic parser type is not actually a monad but a monomorphic linear function `ParserState → ParserState`, avoiding
constructing and deconstructing countless monadic return values. Instead of explicitly returning syntax objects, parsers
push (zero or more of) them onto a syntax stack inside the linear state. Chaining parsers via `>>` accumulates their
output on the stack. Combinators such as `node` then pop off all syntax objects produced during their invocation and
wrap them in a single `Syntax.node` object that is again pushed on this stack. Instead of calling `node` directly, we
usually use the macro `leading_parser p`, which unfolds to `node k p` where the new syntax node kind `k` is the name of the
declaration being defined.

The lack of a dedicated lexer ensures we can modify and replace the lexical grammar at any point, and simplifies
detecting and propagating whitespace. The parser still has a concept of "tokens", however, and caches the most recent
one for performance: when `tokenFn` is called twice at the same position in the input, it will reuse the result of the
```

```
first call. `tokenFn` recognizes some built-in variable-length tokens such as identifiers as well as any fixed token in
the `ParserContext`'s `TokenTable` (a trie); however, the same cache field and strategy could be reused by custom token
parsers. Tokens also play a central role in the `prattParser` combinator, which selects a *leading* parser followed by
zero or more *trailing* parsers based on the current token (via `peekToken`); see the documentation of `prattParser`
for more details. Tokens are specified via the `symbol` parser, or with `symbolNoWs` for tokens that should not be preceded by whitespa

The `Parser` type is extended with additional metadata over the mere parsing function to propagate token information:
`collectTokens` collects all tokens within a parser for registering. `firstTokens` holds information about the "FIRST"
token set used to speed up parser selection in `prattParser`. This approach of combining static and dynamic information
in the parser type is inspired by the paper "Deterministic, Error-Correcting Combinator Parsers" by Swierstra and Duponcheel.
If multiple parsers accept the same current token, `prattParser` tries all of them using the backtracking `longestMatchFn` combinator.
This is the only case where standard parsers might execute arbitrary backtracking. At the moment there is no memoization shared by thes
parallel parsers apart from the first token, though we might change this in the future if the need arises.

Finally, error reporting follows the standard combinatoric approach of collecting a single unexpected token/... and zero
or more expected tokens (see `Error` below). Expected tokens are e.g. set by `symbol` and merged by `<|>`. Combinators
running multiple parsers should check if an error message is set in the parser state (`hasError`) and act accordingly.
Error recovery is left to the designer of the specific language; for example, Lean's top-level `parseCommand` loop skips
tokens until the next command keyword on error.
-/
import Lean.Data.Trie
import Lean.Data.Position
import Lean.Syntax
import Lean.ToExpr
import Lean.Environment
import Lean.Attributes
import Lean.Message
import Lean.Compiler.InitAttr
import Lean.ResolveName

namespace Lean

namespace Parser

def isLitKind (k : SyntaxNodeKind) : Bool :=
  k == strLitKind || k == numLitKind || k == charLitKind || k == nameLitKind || k == scientificLitKind

abbrev mkAtom (info : SourceInfo) (val : String) : Syntax :=
  Syntax.atom info val

abbrev mkIdent (info : SourceInfo) (rawVal : Substring) (val : Name) : Syntax :=
  Syntax.ident info rawVal val []

/- Return character after position `pos` -/
def getNext (input : String) (pos : Nat) : Char :=
  input.get (input.next pos)

```

```
/- Maximal (and function application) precedence.
   In the standard lean language, no parser has precedence higher than `maxPrec`.

   Note that nothing prevents users from using a higher precedence, but we strongly
   discourage them from doing it. -/
def maxPrec  : Nat := eval_prec max
def argPrec  : Nat := eval_prec arg
def leadPrec : Nat := eval_prec lead
def minPrec  : Nat := eval_prec min

abbrev Token := String

structure TokenCacheEntry where
  startPos : String.Pos := 0
  stopPos  : String.Pos := 0
  token    : Syntax := Syntax.missing

structure ParserCache where
  tokenCache : TokenCacheEntry

def initCacheForInput (input : String) : ParserCache := {
  tokenCache := { startPos := input.bsize + 1 /- make sure it is not a valid position -/}
}

abbrev TokenTable := Trie Token

abbrev SyntaxNodeKindSet := Std.PersistentHashMap SyntaxNodeKind Unit

def SyntaxNodeKindSet.insert (s : SyntaxNodeKindSet) (k : SyntaxNodeKind) : SyntaxNodeKindSet :=
  Std.PersistentHashMap.insert s k ()

/-
  Input string and related data. Recall that the `FileMap` is a helper structure for mapping
  `String.Pos` in the input string to line/column information.  -/
structure InputContext where
  input    : String
  fileName : String
  fileMap  : FileMap
  deriving Inhabited

/-- Input context derived from elaboration of previous commands. -/
structure ParserModuleContext where
  env           : Environment
  options       : Options
  -- for name lookup
  currNamespace : Name := Name.anonymous
  openDecls     : List OpenDecl := []
```

```lean

structure ParserContext extends InputContext, ParserModuleContext where
  prec               : Nat
  tokens             : TokenTable
  insideQuot         : Bool := false
  suppressInsideQuot : Bool := false
  savedPos?          : Option String.Pos := none
  forbiddenTk?       : Option Token := none

def ParserContext.resolveName (ctx : ParserContext) (id : Name) : List (Name × List String) :=
  ResolveName.resolveGlobalName ctx.env ctx.currNamespace ctx.openDecls id

structure Error where
  unexpected : String := ""
  expected : List String := []
  deriving Inhabited, BEq

namespace Error

private def expectedToString : List String → String
  | []       => ""
  | [e]      => e
  | [e1, e2] => e1 ++ " or " ++ e2
  | e::es    => e ++ ", " ++ expectedToString es

protected def toString (e : Error) : String :=
  let unexpected := if e.unexpected == "" then [] else [e.unexpected]
  let expected   := if e.expected == [] then [] else
    let expected := e.expected.toArray.qsort (fun e e' => e < e')
    let expected := expected.toList.eraseReps
    ["expected " ++ expectedToString expected]
  "; ".intercalate $ unexpected ++ expected

instance : ToString Error := ⟨Error.toString⟩

def merge (e₁ e₂ : Error) : Error :=
  match e₂ with
  | { unexpected := u, .. } => { unexpected := if u == "" then e₁.unexpected else u, expected := e₁.expected ++ e₂.expected }

end Error

structure ParserState where
  stxStack : Array Syntax := #[]
  /--
    Set to the precedence of the preceding (not surrounding) parser by `runLongestMatchParser`
    for the use of `checkLhsPrec` in trailing parsers.
    Note that with chaining, the preceding parser can be another trailing parser:
```

```
    in `1 * 2 + 3`, the preceding parser is '*' when '+' is executed. -/
  lhsPrec  : Nat := 0
  pos      : String.Pos := 0
  cache    : ParserCache
  errorMsg : Option Error := none

namespace ParserState

@[inline] def hasError (s : ParserState) : Bool :=
  s.errorMsg != none

@[inline] def stackSize (s : ParserState) : Nat :=
  s.stxStack.size

def restore (s : ParserState) (iniStackSz : Nat) (iniPos : Nat) : ParserState :=
  { s with stxStack := s.stxStack.shrink iniStackSz, errorMsg := none, pos := iniPos }

def setPos (s : ParserState) (pos : Nat) : ParserState :=
  { s with pos := pos }

def setCache (s : ParserState) (cache : ParserCache) : ParserState :=
  { s with cache := cache }

def pushSyntax (s : ParserState) (n : Syntax) : ParserState :=
  { s with stxStack := s.stxStack.push n }

def popSyntax (s : ParserState) : ParserState :=
  { s with stxStack := s.stxStack.pop }

def shrinkStack (s : ParserState) (iniStackSz : Nat) : ParserState :=
  { s with stxStack := s.stxStack.shrink iniStackSz }

def next (s : ParserState) (input : String) (pos : Nat) : ParserState :=
  { s with pos := input.next pos }

def toErrorMsg (ctx : ParserContext) (s : ParserState) : String :=
  match s.errorMsg with
  | none     => ""
  | some msg =>
    let pos := ctx.fileMap.toPosition s.pos
    mkErrorStringWithPos ctx.fileName pos (toString msg)

def mkNode (s : ParserState) (k : SyntaxNodeKind) (iniStackSz : Nat) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, err⟩ =>
    if err != none && stack.size == iniStackSz then
      -- If there is an error but there are no new nodes on the stack, use `missing` instead.
```

```
    -- Thus we ensure the property that an syntax tree contains (at least) one `missing` node
    -- if (and only if) there was a parse error.
    -- We should not create an actual node of kind `k` in this case because it would mean we
    -- choose an "arbitrary" node (in practice the last one) in an alternative of the form
    -- `node k1 p1 <|> ... <|> node kn pn` when all parsers fail. With the code below we
    -- instead return a less misleading single `missing` node without randomly selecting any `ki`.
    let stack   := stack.push Syntax.missing
    ⟨stack, lhsPrec, pos, cache, err⟩
  else
    let newNode := Syntax.node k (stack.extract iniStackSz stack.size)
    let stack   := stack.shrink iniStackSz
    let stack   := stack.push newNode
    ⟨stack, lhsPrec, pos, cache, err⟩

def mkTrailingNode (s : ParserState) (k : SyntaxNodeKind) (iniStackSz : Nat) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, err⟩ =>
    let newNode := Syntax.node k (stack.extract (iniStackSz - 1) stack.size)
    let stack   := stack.shrink (iniStackSz - 1)
    let stack   := stack.push newNode
    ⟨stack, lhsPrec, pos, cache, err⟩

def mkError (s : ParserState) (msg : String) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, _⟩ => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ] }⟩

def mkUnexpectedError (s : ParserState) (msg : String) (expected : List String := []) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, _⟩ => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg, expected := expected

def mkEOIError (s : ParserState) (expected : List String := []) : ParserState :=
  s.mkUnexpectedError "unexpected end of input" expected

def mkErrorAt (s : ParserState) (msg : String) (pos : String.Pos) (initStackSz? : Option Nat := none) : ParserState :=
  match s,  initStackSz? with
  | ⟨stack, lhsPrec, _, cache, _⟩, none    => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ] }⟩
  | ⟨stack, lhsPrec, _, cache, _⟩, some sz => ⟨stack.shrink sz |>.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ]

def mkErrorsAt (s : ParserState) (ex : List String) (pos : String.Pos) (initStackSz? : Option Nat := none) : ParserState :=
  match s, initStackSz? with
  | ⟨stack, lhsPrec, _, cache, _⟩, none    => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := ex }⟩
  | ⟨stack, lhsPrec, _, cache, _⟩, some sz => ⟨stack.shrink sz |>.push Syntax.missing, lhsPrec, pos, cache, some { expected := ex }⟩

def mkUnexpectedErrorAt (s : ParserState) (msg : String) (pos : String.Pos) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, _, cache, _⟩ => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg }⟩
```

```
end ParserState

def ParserFn := ParserContext → ParserState → ParserState

instance : Inhabited ParserFn where
  default := fun ctx s => s

inductive FirstTokens where
  | epsilon   : FirstTokens
  | unknown   : FirstTokens
  | tokens    : List Token → FirstTokens
  | optTokens : List Token → FirstTokens
  deriving Inhabited

namespace FirstTokens

def seq : FirstTokens → FirstTokens → FirstTokens
  | epsilon,      tks          => tks
  | optTokens s₁, optTokens s₂ => optTokens (s₁ ++ s₂)
  | optTokens s₁, tokens s₂    => tokens (s₁ ++ s₂)
  | tks,          _            => tks

def toOptional : FirstTokens → FirstTokens
  | tokens tks => optTokens tks
  | tks        => tks

def merge : FirstTokens → FirstTokens → FirstTokens
  | epsilon,      tks          => toOptional tks
  | tks,          epsilon      => toOptional tks
  | tokens s₁,    tokens s₂    => tokens (s₁ ++ s₂)
  | optTokens s₁, optTokens s₂ => optTokens (s₁ ++ s₂)
  | tokens s₁,    optTokens s₂ => optTokens (s₁ ++ s₂)
  | optTokens s₁, tokens s₂    => optTokens (s₁ ++ s₂)
  | _,            _            => unknown

def toStr : FirstTokens → String
  | epsilon       => "epsilon"
  | unknown       => "unknown"
  | tokens tks    => toString tks
  | optTokens tks => "?" ++ toString tks

instance : ToString FirstTokens := ⟨toStr⟩

end FirstTokens

structure ParserInfo where
  collectTokens : List Token → List Token := id
```

```
  collectKinds  : SyntaxNodeKindSet → SyntaxNodeKindSet := id
  firstTokens   : FirstTokens := FirstTokens.unknown
  deriving Inhabited

structure Parser where
  info : ParserInfo := {}
  fn   : ParserFn
  deriving Inhabited

abbrev TrailingParser := Parser

def dbgTraceStateFn (label : String) (p : ParserFn) : ParserFn :=
  fun c s =>
    let sz := s.stxStack.size
    let s' := p c s
    dbg_trace "{label}
  pos: {s'.pos}
  err: {s'.errorMsg}
  out: {s'.stxStack.extract sz s'.stxStack.size}" s'

def dbgTraceState (label : String) (p : Parser) : Parser where
  fn   := dbgTraceStateFn label p.fn
  info := p.info

@[noinline] def epsilonInfo : ParserInfo :=
  { firstTokens := FirstTokens.epsilon }

@[inline] def checkStackTopFn (p : Syntax → Bool) (msg : String) : ParserFn := fun c s =>
  if p s.stxStack.back then s
  else s.mkUnexpectedError msg

@[inline] def checkStackTop (p : Syntax → Bool) (msg : String) : Parser := {
  info := epsilonInfo,
  fn   := checkStackTopFn p msg
}

@[inline] def andthenFn (p q : ParserFn) : ParserFn := fun c s =>
  let s := p c s
  if s.hasError then s else q c s

@[noinline] def andthenInfo (p q : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens ∘ q.collectTokens,
  collectKinds  := p.collectKinds ∘ q.collectKinds,
  firstTokens   := p.firstTokens.seq q.firstTokens
}

@[inline] def andthen (p q : Parser) : Parser := {
```

```
  info := andthenInfo p.info q.info,
  fn   := andthenFn p.fn q.fn
}

instance : AndThen Parser := ⟨andthen⟩

@[inline] def nodeFn (n : SyntaxNodeKind) (p : ParserFn) : ParserFn := fun c s =>
  let iniSz := s.stackSize
  let s     := p c s
  s.mkNode n iniSz

@[inline] def trailingNodeFn (n : SyntaxNodeKind) (p : ParserFn) : ParserFn := fun c s =>
  let iniSz := s.stackSize
  let s     := p c s
  s.mkTrailingNode n iniSz

@[noinline] def nodeInfo (n : SyntaxNodeKind) (p : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens,
  collectKinds  := fun s => (p.collectKinds s).insert n,
  firstTokens   := p.firstTokens
}

@[inline] def node (n : SyntaxNodeKind) (p : Parser) : Parser := {
  info := nodeInfo n p.info,
  fn   := nodeFn n p.fn
}

def errorFn (msg : String) : ParserFn := fun _ s =>
  s.mkUnexpectedError msg

@[inline] def error (msg : String) : Parser := {
  info := epsilonInfo,
  fn   := errorFn msg
}

def errorAtSavedPosFn (msg : String) (delta : Bool) : ParserFn := fun c s =>
  match c.savedPos? with
  | none     => s
  | some pos =>
    let pos := if delta then c.input.next pos else pos
    match s with
    | ⟨stack, lhsPrec, _, cache, _⟩ => ⟨stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg }⟩

/- Generate an error at the position saved with the `withPosition` combinator.
   If `delta == true`, then it reports at saved position+1.
   This useful to make sure a parser consumed at least one character.  -/
@[inline] def errorAtSavedPos (msg : String) (delta : Bool) : Parser := {
```

```
  fn := errorAtSavedPosFn msg delta
}

/- Succeeds if `c.prec <= prec` -/
def checkPrecFn (prec : Nat) : ParserFn := fun c s =>
  if c.prec <= prec then s
  else s.mkUnexpectedError "unexpected token at this precedence level; consider parenthesizing the term"

@[inline] def checkPrec (prec : Nat) : Parser := {
  info := epsilonInfo,
  fn   := checkPrecFn prec
}

/- Succeeds if `c.lhsPrec >= prec` -/
def checkLhsPrecFn (prec : Nat) : ParserFn := fun c s =>
  if s.lhsPrec >= prec then s
  else s.mkUnexpectedError "unexpected token at this precedence level; consider parenthesizing the term"

@[inline] def checkLhsPrec (prec : Nat) : Parser := {
  info := epsilonInfo,
  fn   := checkLhsPrecFn prec
}

def setLhsPrecFn (prec : Nat) : ParserFn := fun c s =>
  if s.hasError then s
  else { s with lhsPrec := prec }

@[inline] def setLhsPrec (prec : Nat) : Parser := {
  info := epsilonInfo,
  fn   := setLhsPrecFn prec
}

def checkInsideQuotFn : ParserFn := fun c s =>
  if c.insideQuot then s
  else s.mkUnexpectedError "unexpected syntax outside syntax quotation"

@[inline] def checkInsideQuot : Parser := {
  info := epsilonInfo,
  fn   := checkInsideQuotFn
}

def checkOutsideQuotFn : ParserFn := fun c s =>
  if !c.insideQuot then s
  else s.mkUnexpectedError "unexpected syntax inside syntax quotation"

@[inline] def checkOutsideQuot : Parser := {
  info := epsilonInfo,
```

```
  fn   := checkOutsideQuotFn
}

def toggleInsideQuotFn (p : ParserFn) : ParserFn := fun c s =>
  if c.suppressInsideQuot then p c s
  else p { c with insideQuot := !c.insideQuot } s

@[inline] def toggleInsideQuot (p : Parser) : Parser := {
  info := p.info,
  fn   := toggleInsideQuotFn p.fn
}

def suppressInsideQuotFn (p : ParserFn) : ParserFn := fun c s =>
  p { c with suppressInsideQuot := true } s

@[inline] def suppressInsideQuot (p : Parser) : Parser := {
  info := p.info,
  fn   := suppressInsideQuotFn p.fn
}

@[inline] def leadingNode (n : SyntaxNodeKind) (prec : Nat) (p : Parser) : Parser :=
  checkPrec prec >> node n p >> setLhsPrec prec

@[inline] def trailingNodeAux (n : SyntaxNodeKind) (p : Parser) : TrailingParser := {
  info := nodeInfo n p.info,
  fn   := trailingNodeFn n p.fn
}

@[inline] def trailingNode (n : SyntaxNodeKind) (prec lhsPrec : Nat) (p : Parser) : TrailingParser :=
  checkPrec prec >> checkLhsPrec lhsPrec >> trailingNodeAux n p >> setLhsPrec prec

def mergeOrElseErrors (s : ParserState) (error1 : Error) (iniPos : Nat) (mergeErrors : Bool) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, some error2⟩ =>
    if pos == iniPos then ⟨stack, lhsPrec, pos, cache, some (if mergeErrors then error1.merge error2 else error2)⟩
    else s
  | other => other

def orelseFnCore (p q : ParserFn) (mergeErrors : Bool) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := p c s
  match s.errorMsg with
  | some errorMsg =>
    if s.pos == iniPos then
      mergeOrElseErrors (q c (s.restore iniSz iniPos)) errorMsg iniPos mergeErrors
    else
```

```
    s
  | none => s

@[inline] def orelseFn (p q : ParserFn) : ParserFn :=
  orelseFnCore p q true

@[noinline] def orelseInfo (p q : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens ∘ q.collectTokens,
  collectKinds  := p.collectKinds ∘ q.collectKinds,
  firstTokens   := p.firstTokens.merge q.firstTokens
}

/--
  Run `p`, falling back to `q` if `p` failed without consuming any input.

  NOTE: In order for the pretty printer to retrace an `orelse`, `p` must be a call to `node` or some other parser
  producing a single node kind. Nested `orelse` calls are flattened for this, i.e. `(node k1 p1 <|> node k2 p2) <|> ...`
  is fine as well. -/
@[inline] def orelse (p q : Parser) : Parser := {
  info := orelseInfo p.info q.info,
  fn   := orelseFn p.fn q.fn
}

instance : OrElse Parser := ⟨orelse⟩

@[noinline] def noFirstTokenInfo (info : ParserInfo) : ParserInfo := {
  collectTokens := info.collectTokens,
  collectKinds  := info.collectKinds
}

def atomicFn (p : ParserFn) : ParserFn := fun c s =>
  let iniPos := s.pos
  match p c s with
  | ⟨stack, lhsPrec, _, cache, some msg⟩ => ⟨stack, lhsPrec, iniPos, cache, some msg⟩
  | other                       => other

@[inline] def atomic (p : Parser) : Parser := {
  info := p.info,
  fn   := atomicFn p.fn
}

def optionalFn (p : ParserFn) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := p c s
  let s      := if s.hasError && s.pos == iniPos then s.restore iniSz iniPos else s
  s.mkNode nullKind iniSz
```

```
@[noinline] def optionaInfo (p : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens,
  collectKinds  := p.collectKinds,
  firstTokens   := p.firstTokens.toOptional
}

@[inline] def optionalNoAntiquot (p : Parser) : Parser := {
  info := optionaInfo p.info,
  fn   := optionalFn p.fn
}

def lookaheadFn (p : ParserFn) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := p c s
  if s.hasError then s else s.restore iniSz iniPos

@[inline] def lookahead (p : Parser) : Parser := {
  info := p.info,
  fn   := lookaheadFn p.fn
}

def notFollowedByFn (p : ParserFn) (msg : String) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := p c s
  if s.hasError then
    s.restore iniSz iniPos
  else
    let s := s.restore iniSz iniPos
    s.mkUnexpectedError s!"unexpected {msg}"

@[inline] def notFollowedBy (p : Parser) (msg : String) : Parser := {
  fn := notFollowedByFn p.fn msg
}

partial def manyAux (p : ParserFn) : ParserFn := fun c s => do
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let mut s  := p c s
  if s.hasError then
    return if iniPos == s.pos then s.restore iniSz iniPos else s
  if iniPos == s.pos then
    return s.mkUnexpectedError "invalid 'many' parser combinator application, parser did not consume anything"
  if s.stackSize > iniSz + 1 then
    s := s.mkNode nullKind iniSz
```

```
  manyAux p c s

@[inline] def manyFn (p : ParserFn) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let s := manyAux p c s
  s.mkNode nullKind iniSz

@[inline] def manyNoAntiquot (p : Parser) : Parser := {
  info := noFirstTokenInfo p.info,
  fn   := manyFn p.fn
}

@[inline] def many1Fn (p : ParserFn) : ParserFn := fun c s =>
  let iniSz  := s.stackSize
  let s := andthenFn p (manyAux p) c s
  s.mkNode nullKind iniSz

@[inline] def many1NoAntiquot (p : Parser) : Parser := {
  info := p.info,
  fn   := many1Fn p.fn
}

private partial def sepByFnAux (p : ParserFn) (sep : ParserFn) (allowTrailingSep : Bool) (iniSz : Nat) (pOpt : Bool) : ParserFn :=
  let rec parse (pOpt : Bool) (c s) := do
    let sz  := s.stackSize
    let pos := s.pos
    let mut s := p c s
    if s.hasError then
      if s.pos > pos then
        return s.mkNode nullKind iniSz
      else if pOpt then
        let s := s.restore sz pos
        return s.mkNode nullKind iniSz
      else
        -- append `Syntax.missing` to make clear that List is incomplete
        let s := s.pushSyntax Syntax.missing
        return s.mkNode nullKind iniSz
    if s.stackSize > sz + 1 then
      s := s.mkNode nullKind sz
    let sz  := s.stackSize
    let pos := s.pos
    let s   := sep c s
    if s.hasError then
      let s := s.restore sz pos
      return s.mkNode nullKind iniSz
    if s.stackSize > sz + 1 then
      s := s.mkNode nullKind sz
```

```
    parse allowTrailingSep c s
  parse pOpt

def sepByFn (allowTrailingSep : Bool) (p : ParserFn) (sep : ParserFn) : ParserFn := fun c s =>
  let iniSz := s.stackSize
  sepByFnAux p sep allowTrailingSep iniSz true c s

def sepBy1Fn (allowTrailingSep : Bool) (p : ParserFn) (sep : ParserFn) : ParserFn := fun c s =>
  let iniSz := s.stackSize
  sepByFnAux p sep allowTrailingSep iniSz false c s

@[noinline] def sepByInfo (p sep : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens ∘ sep.collectTokens,
  collectKinds  := p.collectKinds ∘ sep.collectKinds
}

@[noinline] def sepBy1Info (p sep : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens ∘ sep.collectTokens,
  collectKinds  := p.collectKinds ∘ sep.collectKinds,
  firstTokens   := p.firstTokens
}

@[inline] def sepByNoAntiquot (p sep : Parser) (allowTrailingSep : Bool := false) : Parser := {
  info := sepByInfo p.info sep.info,
  fn   := sepByFn allowTrailingSep p.fn sep.fn
}

@[inline] def sepBy1NoAntiquot (p sep : Parser) (allowTrailingSep : Bool := false) : Parser := {
  info := sepBy1Info p.info sep.info,
  fn   := sepBy1Fn allowTrailingSep p.fn sep.fn
}

/- Apply `f` to the syntax object produced by `p` -/
def withResultOfFn (p : ParserFn) (f : Syntax → Syntax) : ParserFn := fun c s =>
  let s := p c s
  if s.hasError then s
  else
    let stx := s.stxStack.back
    s.popSyntax.pushSyntax (f stx)

@[noinline] def withResultOfInfo (p : ParserInfo) : ParserInfo := {
  collectTokens := p.collectTokens,
  collectKinds  := p.collectKinds
}

@[inline] def withResultOf (p : Parser) (f : Syntax → Syntax) : Parser := {
  info := withResultOfInfo p.info,
```

```
  fn   := withResultOfFn p.fn f
}

@[inline] def many1Unbox (p : Parser) : Parser :=
  withResultOf (many1NoAntiquot p) fun stx => if stx.getNumArgs == 1 then stx.getArg 0 else stx

partial def satisfyFn (p : Char → Bool) (errorMsg : String := "unexpected character") : ParserFn := fun c s =>
  let i := s.pos
  if c.input.atEnd i then s.mkEOIError
  else if p (c.input.get i) then s.next c.input i
  else s.mkUnexpectedError errorMsg

partial def takeUntilFn (p : Char → Bool) : ParserFn := fun c s =>
  let i := s.pos
  if c.input.atEnd i then s
  else if p (c.input.get i) then s
  else takeUntilFn p c (s.next c.input i)

def takeWhileFn (p : Char → Bool) : ParserFn :=
  takeUntilFn (fun c => !p c)

@[inline] def takeWhile1Fn (p : Char → Bool) (errorMsg : String) : ParserFn :=
  andthenFn (satisfyFn p errorMsg) (takeWhileFn p)

variable (startPos : String.Pos) in
partial def finishCommentBlock (nesting : Nat) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then eoi s
  else
    let curr := input.get i
    let i    := input.next i
    if curr == '-' then
      if input.atEnd i then eoi s
      else
        let curr := input.get i
        if curr == '/' then -- "-/" end of comment
          if nesting == 1 then s.next input i
          else finishCommentBlock (nesting-1) c (s.next input i)
        else
          finishCommentBlock nesting c (s.next input i)
    else if curr == '/' then
      if input.atEnd i then eoi s
      else
        let curr := input.get i
        if curr == '-' then finishCommentBlock (nesting+1) c (s.next input i)
        else finishCommentBlock nesting c (s.setPos i)
```

```
    else finishCommentBlock nesting c (s.setPos i)
where
  eoi s := s.mkUnexpectedErrorAt "unterminated comment" startPos

/- Consume whitespace and comments -/
partial def whitespace : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s
  else
    let curr := input.get i
    if curr.isWhitespace then whitespace c (s.next input i)
    else if curr == '-' then
      let i    := input.next i
      let curr := input.get i
      if curr == '-' then andthenFn (takeUntilFn (fun c => c = '\n')) whitespace c (s.next input i)
      else s
    else if curr == '/' then
      let startPos := i
      let i        := input.next i
      let curr     := input.get i
      if curr == '-' then
        let i    := input.next i
        let curr := input.get i
        if curr == '-' then s -- "/--" doc comment is an actual token
        else andthenFn (finishCommentBlock startPos 1) whitespace c (s.next input i)
      else s
    else s

def mkEmptySubstringAt (s : String) (p : Nat) : Substring :=
  { str := s, startPos := p, stopPos := p }

private def rawAux (startPos : Nat) (trailingWs : Bool) : ParserFn := fun c s =>
  let input   := c.input
  let stopPos := s.pos
  let leading := mkEmptySubstringAt input startPos
  let val     := input.extract startPos stopPos
  if trailingWs then
    let s        := whitespace c s
    let stopPos' := s.pos
    let trailing := { str := input, startPos := stopPos, stopPos := stopPos' : Substring }
    let atom     := mkAtom (SourceInfo.original leading startPos trailing) val
    s.pushSyntax atom
  else
    let trailing := mkEmptySubstringAt input stopPos
    let atom     := mkAtom (SourceInfo.original leading startPos trailing) val
    s.pushSyntax atom
```

```
/-- Match an arbitrary Parser and return the consumed String in a `Syntax.atom`. -/
@[inline] def rawFn (p : ParserFn) (trailingWs := false) : ParserFn := fun c s =>
  let startPos := s.pos
  let s := p c s
  if s.hasError then s else rawAux startPos trailingWs c s

@[inline] def chFn (c : Char) (trailingWs := false) : ParserFn :=
  rawFn (satisfyFn (fun d => c == d) ("'" ++ toString c ++ "'")) trailingWs

def rawCh (c : Char) (trailingWs := false) : Parser :=
  { fn := chFn c trailingWs }

def hexDigitFn : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError
  else
    let curr := input.get i
    let i    := input.next i
    if curr.isDigit || ('a' <= curr && curr <= 'f') || ('A' <= curr && curr <= 'F') then s.setPos i
    else s.mkUnexpectedError "invalid hexadecimal numeral"

def quotedCharCoreFn (isQuotable : Char → Bool) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError
  else
    let curr := input.get i
    if isQuotable curr then
      s.next input i
    else if curr == 'x' then
      andthenFn hexDigitFn hexDigitFn c (s.next input i)
    else if curr == 'u' then
      andthenFn hexDigitFn (andthenFn hexDigitFn (andthenFn hexDigitFn hexDigitFn)) c (s.next input i)
    else
      s.mkUnexpectedError "invalid escape sequence"

def isQuotableCharDefault (c : Char) : Bool :=
  c == '\\' || c == '\"' || c == '\'' || c == 'r' || c == 'n' || c == 't'

def quotedCharFn : ParserFn :=
  quotedCharCoreFn isQuotableCharDefault

/-- Push `(Syntax.node tk <new-atom>)` into syntax stack -/
def mkNodeToken (n : SyntaxNodeKind) (startPos : Nat) : ParserFn := fun c s =>
  let input     := c.input
```

```
  let stopPos   := s.pos
  let leading   := mkEmptySubstringAt input startPos
  let val       := input.extract startPos stopPos
  let s         := whitespace c s
  let wsStopPos := s.pos
  let trailing  := { str := input, startPos := stopPos, stopPos := wsStopPos : Substring }
  let info      := SourceInfo.original leading startPos trailing
  s.pushSyntax (Syntax.mkLit n val info)

def charLitFnAux (startPos : Nat) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError
  else
    let curr := input.get i
    let s    := s.setPos (input.next i)
    let s    := if curr == '\\' then quotedCharFn c s else s
    if s.hasError then s
    else
      let i    := s.pos
      let curr := input.get i
      let s    := s.setPos (input.next i)
      if curr == '\'' then mkNodeToken charLitKind startPos c s
      else s.mkUnexpectedError "missing end of character literal"

partial def strLitFnAux (startPos : Nat) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkUnexpectedErrorAt "unterminated string literal" startPos
  else
    let curr := input.get i
    let s    := s.setPos (input.next i)
    if curr == '\"' then
      mkNodeToken strLitKind startPos c s
    else if curr == '\\' then andthenFn quotedCharFn (strLitFnAux startPos) c s
    else strLitFnAux startPos c s

def decimalNumberFn (startPos : Nat) (c : ParserContext) : ParserState → ParserState := fun s =>
  let s     := takeWhileFn (fun c => c.isDigit) c s
  let input := c.input
  let i     := s.pos
  let curr  := input.get i
  if curr == '.' || curr == 'e' || curr == 'E' then
    let s := parseOptDot s
    let s := parseOptExp s
    mkNodeToken scientificLitKind startPos c s
  else
```

```
    mkNodeToken numLitKind startPos c s
where
  parseOptDot s :=
    let input := c.input
    let i     := s.pos
    let curr  := input.get i
    if curr == '.' then
      let i    := input.next i
      let curr := input.get i
      if curr.isDigit then
        takeWhileFn (fun c => c.isDigit) c (s.setPos i)
      else
        s.setPos i
    else
      s

  parseOptExp s :=
    let input := c.input
    let i     := s.pos
    let curr  := input.get i
    if curr == 'e' || curr == 'E' then
      let i    := input.next i
      let i    := if input.get i == '-' then input.next i else i
      let curr := input.get i
      if curr.isDigit then
        takeWhileFn (fun c => c.isDigit) c (s.setPos i)
      else
        s.setPos i
    else
      s

def binNumberFn (startPos : Nat) : ParserFn := fun c s =>
  let s := takeWhile1Fn (fun c => c == '0' || c == '1') "binary number" c s
  mkNodeToken numLitKind startPos c s

def octalNumberFn (startPos : Nat) : ParserFn := fun c s =>
  let s := takeWhile1Fn (fun c => '0' ≤ c && c ≤ '7') "octal number" c s
  mkNodeToken numLitKind startPos c s

def hexNumberFn (startPos : Nat) : ParserFn := fun c s =>
  let s := takeWhile1Fn (fun c => ('0' ≤ c && c ≤ '9') || ('a' ≤ c && c ≤ 'f') || ('A' ≤ c && c ≤ 'F')) "hexadecimal number" c s
  mkNodeToken numLitKind startPos c s

def numberFnAux : ParserFn := fun c s =>
  let input    := c.input
  let startPos := s.pos
  if input.atEnd startPos then s.mkEOIError
```

```
  else
    let curr := input.get startPos
    if curr == '0' then
      let i    := input.next startPos
      let curr := input.get i
      if curr == 'b' || curr == 'B' then
        binNumberFn startPos c (s.next input i)
      else if curr == 'o' || curr == 'O' then
        octalNumberFn startPos c (s.next input i)
      else if curr == 'x' || curr == 'X' then
        hexNumberFn startPos c (s.next input i)
      else
        decimalNumberFn startPos c (s.setPos i)
    else if curr.isDigit then
      decimalNumberFn startPos c (s.next input startPos)
    else
      s.mkError "numeral"

def isIdCont : String → ParserState → Bool := fun input s =>
  let i    := s.pos
  let curr := input.get i
  if curr == '.' then
    let i := input.next i
    if input.atEnd i then
      false
    else
      let curr := input.get i
      isIdFirst curr || isIdBeginEscape curr
  else
    false

private def isToken (idStartPos idStopPos : Nat) (tk : Option Token) : Bool :=
  match tk with
  | none    => false
  | some tk =>
     -- if a token is both a symbol and a valid identifier (i.e. a keyword),
     -- we want it to be recognized as a symbol
    tk.bsize ≥ idStopPos - idStartPos


def mkTokenAndFixPos (startPos : Nat) (tk : Option Token) : ParserFn := fun c s =>
  match tk with
  | none    => s.mkErrorAt "token" startPos
  | some tk =>
    if c.forbiddenTk? == some tk then
      s.mkErrorAt "forbidden token" startPos
    else
```

```
    let input     := c.input
    let leading   := mkEmptySubstringAt input startPos
    let stopPos   := startPos + tk.bsize
    let s         := s.setPos stopPos
    let s         := whitespace c s
    let wsStopPos := s.pos
    let trailing  := { str := input, startPos := stopPos, stopPos := wsStopPos : Substring }
    let atom      := mkAtom (SourceInfo.original leading startPos trailing) tk
    s.pushSyntax atom

def mkIdResult (startPos : Nat) (tk : Option Token) (val : Name) : ParserFn := fun c s =>
  let stopPos           := s.pos
  if isToken startPos stopPos tk then
    mkTokenAndFixPos startPos tk c s
  else
    let input           := c.input
    let rawVal          := { str := input, startPos := startPos, stopPos := stopPos  : Substring }
    let s               := whitespace c s
    let trailingStopPos := s.pos
    let leading         := mkEmptySubstringAt input startPos
    let trailing        := { str := input, startPos := stopPos, stopPos := trailingStopPos : Substring }
    let info            := SourceInfo.original leading startPos trailing
    let atom            := mkIdent info rawVal val
    s.pushSyntax atom

partial def identFnAux (startPos : Nat) (tk : Option Token) (r : Name) : ParserFn :=
  let rec parse (r : Name) (c s) := do
    let input := c.input
    let i     := s.pos
    if input.atEnd i then
      return s.mkEOIError
    let curr := input.get i
    if isIdBeginEscape curr then
      let startPart := input.next i
      let s         := takeUntilFn isIdEndEscape c (s.setPos startPart)
      if input.atEnd s.pos then
        return s.mkUnexpectedErrorAt "unterminated identifier escape" startPart
      let stopPart  := s.pos
      let s         := s.next c.input s.pos
      let r := Name.mkStr r (input.extract startPart stopPart)
      if isIdCont input s then
        let s := s.next input s.pos
        parse r c s
      else
        mkIdResult startPos tk r c s
    else if isIdFirst curr then
      let startPart := i
```

```
      let s         := takeWhileFn isIdRest c (s.next input i)
      let stopPart  := s.pos
      let r := Name.mkStr r (input.extract startPart stopPart)
      if isIdCont input s then
        let s := s.next input s.pos
        parse r c s
      else
        mkIdResult startPos tk r c s
    else
      mkTokenAndFixPos startPos tk c s
  parse r

private def isIdFirstOrBeginEscape (c : Char) : Bool :=
  isIdFirst c || isIdBeginEscape c

private def nameLitAux (startPos : Nat) : ParserFn := fun c s =>
  let input := c.input
  let s     := identFnAux startPos none Name.anonymous c (s.next input startPos)
  if s.hasError then
    s
  else
    let stx := s.stxStack.back
    match stx with
    | Syntax.ident _ rawStr _ _ =>
      let s := s.popSyntax
      s.pushSyntax (Syntax.node nameLitKind #[mkAtomFrom stx rawStr.toString])
    | _ => s.mkError "invalid Name literal"

private def tokenFnAux : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  let curr  := input.get i
  if curr == '\"' then
    strLitFnAux i c (s.next input i)
  else if curr == '\'' then
    charLitFnAux i c (s.next input i)
  else if curr.isDigit then
    numberFnAux c s
  else if curr == '`' && isIdFirstOrBeginEscape (getNext input i) then
    nameLitAux i c s
  else
    let (_, tk) := c.tokens.matchPrefix input i
    identFnAux i tk Name.anonymous c s

private def updateCache (startPos : Nat) (s : ParserState) : ParserState :=
  -- do not cache token parsing errors, which are rare and usually fatal and thus not worth an extra field in `TokenCache`
  match s with
```

```
  | ⟨stack, lhsPrec, pos, cache, none⟩ =>
    if stack.size == 0 then s
    else
      let tk := stack.back
      ⟨stack, lhsPrec, pos, { tokenCache := { startPos := startPos, stopPos := pos, token := tk } }, none⟩
  | other => other

def tokenFn (expected : List String := []) : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError expected
  else
    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

def peekTokenAux (c : ParserContext) (s : ParserState) : ParserState × Except ParserState Syntax :=
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := tokenFn [] c s
  if let some e := s.errorMsg then (s.restore iniSz iniPos, Except.error s)
  else
    let stx := s.stxStack.back
    (s.restore iniSz iniPos, Except.ok stx)

def peekToken (c : ParserContext) (s : ParserState) : ParserState × Except ParserState Syntax :=
  let tkc := s.cache.tokenCache
  if tkc.startPos == s.pos then
    (s, Except.ok tkc.token)
  else
    peekTokenAux c s

/- Treat keywords as identifiers. -/
def rawIdentFn : ParserFn := fun c s =>
  let input := c.input
  let i     := s.pos
  if input.atEnd i then s.mkEOIError
  else identFnAux i none Name.anonymous c s

@[inline] def satisfySymbolFn (p : String → Bool) (expected : List String) : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let startPos := s.pos
  let s        := tokenFn expected c s
```

```
  if s.hasError then
    s
  else
    match s.stxStack.back with
    | Syntax.atom _ sym => if p sym then s else s.mkErrorsAt expected startPos initStackSz
    | _                 => s.mkErrorsAt expected startPos initStackSz

def symbolFnAux (sym : String) (errorMsg : String) : ParserFn :=
  satisfySymbolFn (fun s => s == sym) [errorMsg]

def symbolInfo (sym : String) : ParserInfo := {
  collectTokens := fun tks => sym :: tks,
  firstTokens   := FirstTokens.tokens [ sym ]
}

@[inline] def symbolFn (sym : String) : ParserFn :=
  symbolFnAux sym ("'" ++ sym ++ "'")

@[inline] def symbolNoAntiquot (sym : String) : Parser :=
  let sym := sym.trim
  { info := symbolInfo sym,
    fn   := symbolFn sym }

def checkTailNoWs (prev : Syntax) : Bool :=
  match prev.getTailInfo with
  | SourceInfo.original _ _ trailing => trailing.stopPos == trailing.startPos
  | _                                => false

/-- Check if the following token is the symbol _or_ identifier `sym`. Useful for
    parsing local tokens that have not been added to the token table (but may have
    been so by some unrelated code).

    For example, the universe `max` Function is parsed using this combinator so that
    it can still be used as an identifier outside of universes (but registering it
    as a token in a Term Syntax would not break the universe Parser). -/
def nonReservedSymbolFnAux (sym : String) (errorMsg : String) : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let startPos := s.pos
  let s := tokenFn [errorMsg] c s
  if s.hasError then s
  else
    match s.stxStack.back with
    | Syntax.atom _ sym' =>
      if sym == sym' then s else s.mkErrorAt errorMsg startPos initStackSz
    | Syntax.ident info rawVal _ _ =>
      if sym == rawVal.toString then
        let s := s.popSyntax
```

```
      s.pushSyntax (Syntax.atom info sym)
    else
      s.mkErrorAt errorMsg startPos initStackSz
  | _ => s.mkErrorAt errorMsg startPos initStackSz

@[inline] def nonReservedSymbolFn (sym : String) : ParserFn :=
  nonReservedSymbolFnAux sym ("'" ++ sym ++ "'")

def nonReservedSymbolInfo (sym : String) (includeIdent : Bool) : ParserInfo := {
  firstTokens  :=
    if includeIdent then
      FirstTokens.tokens [ sym, "ident" ]
    else
      FirstTokens.tokens [ sym ]
}

@[inline] def nonReservedSymbolNoAntiquot (sym : String) (includeIdent := false) : Parser :=
  let sym := sym.trim
  { info := nonReservedSymbolInfo sym includeIdent,
    fn   := nonReservedSymbolFn sym }

partial def strAux (sym : String) (errorMsg : String) (j : Nat) :ParserFn :=
  let rec parse (j c s) :=
    if sym.atEnd j then s
    else
      let i := s.pos
      let input := c.input
      if input.atEnd i || sym.get j != input.get i then s.mkError errorMsg
      else parse (sym.next j) c (s.next input i)
  parse j

def checkTailWs (prev : Syntax) : Bool :=
  match prev.getTailInfo with
  | SourceInfo.original _ _ trailing => trailing.stopPos > trailing.startPos
  | _                                => false

def checkWsBeforeFn (errorMsg : String) : ParserFn := fun c s =>
  let prev := s.stxStack.back
  if checkTailWs prev then s else s.mkError errorMsg

def checkWsBefore (errorMsg : String := "space before") : Parser := {
  info := epsilonInfo,
  fn   := checkWsBeforeFn errorMsg
}

def checkTailLinebreak (prev : Syntax) : Bool :=
  match prev.getTailInfo with
```

```
  | SourceInfo.original _ _ trailing => trailing.contains '\n'
  | _ => false

def checkLinebreakBeforeFn (errorMsg : String) : ParserFn := fun c s =>
  let prev := s.stxStack.back
  if checkTailLinebreak prev then s else s.mkError errorMsg

def checkLinebreakBefore (errorMsg : String := "line break") : Parser := {
  info := epsilonInfo
  fn   := checkLinebreakBeforeFn errorMsg
}

private def pickNonNone (stack : Array Syntax) : Syntax :=
  match stack.findRev? $ fun stx => !stx.isNone with
  | none => Syntax.missing
  | some stx => stx

def checkNoWsBeforeFn (errorMsg : String) : ParserFn := fun c s =>
  let prev := pickNonNone s.stxStack
  if checkTailNoWs prev then s else s.mkError errorMsg

def checkNoWsBefore (errorMsg : String := "no space before") : Parser := {
  info := epsilonInfo,
  fn   := checkNoWsBeforeFn errorMsg
}

def unicodeSymbolFnAux (sym asciiSym : String) (expected : List String) : ParserFn :=
  satisfySymbolFn (fun s => s == sym || s == asciiSym) expected

def unicodeSymbolInfo (sym asciiSym : String) : ParserInfo := {
  collectTokens := fun tks => sym :: asciiSym :: tks,
  firstTokens   := FirstTokens.tokens [ sym, asciiSym ]
}

@[inline] def unicodeSymbolFn (sym asciiSym : String) : ParserFn :=
  unicodeSymbolFnAux sym asciiSym ["'" ++ sym ++ "', '" ++ asciiSym ++ "'"]

@[inline] def unicodeSymbolNoAntiquot (sym asciiSym : String) : Parser :=
  let sym := sym.trim
  let asciiSym := asciiSym.trim
  { info := unicodeSymbolInfo sym asciiSym,
    fn   := unicodeSymbolFn sym asciiSym }

def mkAtomicInfo (k : String) : ParserInfo :=
  { firstTokens := FirstTokens.tokens [ k ] }

def numLitFn : ParserFn :=
```

```
  fun c s =>
    let initStackSz := s.stackSize
    let iniPos := s.pos
    let s      := tokenFn ["numeral"] c s
    if !s.hasError && !(s.stxStack.back.isOfKind numLitKind) then s.mkErrorAt "numeral" iniPos initStackSz else s

@[inline] def numLitNoAntiquot : Parser := {
  fn   := numLitFn,
  info := mkAtomicInfo "numLit"
}

def scientificLitFn : ParserFn :=
  fun c s =>
    let initStackSz := s.stackSize
    let iniPos := s.pos
    let s      := tokenFn ["scientific number"] c s
    if !s.hasError && !(s.stxStack.back.isOfKind scientificLitKind) then s.mkErrorAt "scientific number" iniPos initStackSz else s

@[inline] def scientificLitNoAntiquot : Parser := {
  fn   := scientificLitFn,
  info := mkAtomicInfo "scientificLit"
}

def strLitFn : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let s := tokenFn ["string literal"] c s
  if !s.hasError && !(s.stxStack.back.isOfKind strLitKind) then s.mkErrorAt "string literal" iniPos initStackSz else s

@[inline] def strLitNoAntiquot : Parser := {
  fn   := strLitFn,
  info := mkAtomicInfo "strLit"
}

def charLitFn : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let s := tokenFn ["char literal"] c s
  if !s.hasError && !(s.stxStack.back.isOfKind charLitKind) then s.mkErrorAt "character literal" iniPos initStackSz else s

@[inline] def charLitNoAntiquot : Parser := {
  fn   := charLitFn,
  info := mkAtomicInfo "charLit"
}

def nameLitFn : ParserFn := fun c s =>
  let initStackSz := s.stackSize
```

```
  let iniPos := s.pos
  let s := tokenFn ["Name literal"] c s
  if !s.hasError && !(s.stxStack.back.isOfKind nameLitKind) then s.mkErrorAt "Name literal" iniPos initStackSz else s

@[inline] def nameLitNoAntiquot : Parser := {
  fn   := nameLitFn,
  info := mkAtomicInfo "nameLit"
}

def identFn : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let s      := tokenFn ["identifier"] c s
  if !s.hasError && !(s.stxStack.back.isIdent) then s.mkErrorAt "identifier" iniPos initStackSz else s

@[inline] def identNoAntiquot : Parser := {
  fn   := identFn,
  info := mkAtomicInfo "ident"
}

@[inline] def rawIdentNoAntiquot : Parser := {
  fn := rawIdentFn
}

def identEqFn (id : Name) : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let s      := tokenFn ["identifier"] c s
  if s.hasError then
    s
  else match s.stxStack.back with
    | Syntax.ident _ _ val _ => if val != id then s.mkErrorAt ("expected identifier '" ++ toString id ++ "'") iniPos initStackSz else s
    | _ => s.mkErrorAt "identifier" iniPos initStackSz

@[inline] def identEq (id : Name) : Parser := {
  fn   := identEqFn id,
  info := mkAtomicInfo "ident"
}

namespace ParserState

def keepTop (s : Array Syntax) (startStackSize : Nat) : Array Syntax :=
  let node  := s.back
  s.shrink startStackSize |>.push node

def keepNewError (s : ParserState) (oldStackSize : Nat) : ParserState :=
  match s with
```

```
  | ⟨stack, lhsPrec, pos, cache, err⟩ => ⟨keepTop stack oldStackSize, lhsPrec, pos, cache, err⟩

def keepPrevError (s : ParserState) (oldStackSize : Nat) (oldStopPos : String.Pos) (oldError : Option Error) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, _, cache, _⟩ => ⟨stack.shrink oldStackSize, lhsPrec, oldStopPos, cache, oldError⟩

def mergeErrors (s : ParserState) (oldStackSize : Nat) (oldError : Error) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, some err⟩ =>
    if oldError == err then s
    else ⟨stack.shrink oldStackSize, lhsPrec, pos, cache, some (oldError.merge err)⟩
  | other                         => other

def keepLatest (s : ParserState) (startStackSize : Nat) : ParserState :=
  match s with
  | ⟨stack, lhsPrec, pos, cache, _⟩ => ⟨keepTop stack startStackSize, lhsPrec, pos, cache, none⟩

def replaceLongest (s : ParserState) (startStackSize : Nat) : ParserState :=
  s.keepLatest startStackSize

end ParserState

def invalidLongestMatchParser (s : ParserState) : ParserState :=
  s.mkError "longestMatch parsers must generate exactly one Syntax node"

/--
 Auxiliary function used to execute parsers provided to `longestMatchFn`.
 Push `left?` into the stack if it is not `none`, and execute `p`.

 Remark: `p` must produce exactly one syntax node.
 Remark: the `left?` is not none when we are processing trailing parsers. -/
def runLongestMatchParser (left? : Option Syntax) (startLhsPrec : Nat) (p : ParserFn) : ParserFn := fun c s => do
  /-
    We assume any registered parser `p` has one of two forms:
    * a direct call to `leadingParser` or `trailingParser`
    * a direct call to a (leading) token parser
    In the first case, we can extract the precedence of the parser by having `leadingParser/trailingParser`
    set `ParserState.lhsPrec` to it in the very end so that no nested parser can interfere.
    In the second case, the precedence is effectively `max` (there is a `checkPrec` merely for the convenience
    of the pretty printer) and there are no nested `leadingParser/trailingParser` calls, so the value of `lhsPrec`
    will not be changed by the parser (nor will it be read by any leading parser). Thus we initialize the field
    to `maxPrec` in the leading case. -/
  let mut s := { s with lhsPrec := if left?.isSome then startLhsPrec else maxPrec }
  let startSize := s.stackSize
  if let some left := left? then
    s := s.pushSyntax left
  s := p c s
```

```
  -- stack contains `[..., result ]`
  if s.stackSize == startSize + 1 then
    s -- success or error with the expected number of nodes
  else if s.hasError then
    -- error with an unexpected number of nodes.
    s.shrinkStack startSize |>.pushSyntax Syntax.missing
  else
    -- parser succeded with incorrect number of nodes
    invalidLongestMatchParser s

def longestMatchStep (left? : Option Syntax) (startSize startLhsPrec : Nat) (startPos : String.Pos) (prevPrio : Nat) (prio : Nat) (p :
    : ParserContext → ParserState → ParserState × Nat := fun c s =>
  let prevErrorMsg  := s.errorMsg
  let prevStopPos   := s.pos
  let prevSize      := s.stackSize
  let prevLhsPrec   := s.lhsPrec
  let s             := s.restore prevSize startPos
  let s             := runLongestMatchParser left? startLhsPrec p c s
  match prevErrorMsg, s.errorMsg with
  | none, none   => -- both succeeded
    if s.pos > prevStopPos || (s.pos == prevStopPos && prio > prevPrio)      then (s.replaceLongest startSize, prio)
    else if s.pos < prevStopPos || (s.pos == prevStopPos && prio < prevPrio) then ({ s.restore prevSize prevStopPos with lhsPrec := pre
    -- it is not clear what the precedence of a choice node should be, so we conservatively take the minimum
    else ({s with lhsPrec := s.lhsPrec.min prevLhsPrec }, prio)
  | none, some _ => -- prev succeeded, current failed
    ({ s.restore prevSize prevStopPos with lhsPrec := prevLhsPrec }, prevPrio)
  | some oldError, some _ => -- both failed
    if s.pos > prevStopPos || (s.pos == prevStopPos && prio > prevPrio)      then (s.keepNewError startSize, prio)
    else if s.pos < prevStopPos || (s.pos == prevStopPos && prio < prevPrio) then (s.keepPrevError prevSize prevStopPos prevErrorMsg, p
    else (s.mergeErrors prevSize oldError, prio)
  | some _, none => -- prev failed, current succeeded
    let successNode := s.stxStack.back
    let s           := s.shrinkStack startSize -- restore stack to initial size to make sure (failure) nodes are removed from the stack
    (s.pushSyntax successNode, prio) -- put successNode back on the stack

def longestMatchMkResult (startSize : Nat) (s : ParserState) : ParserState :=
  if !s.hasError && s.stackSize > startSize + 1 then s.mkNode choiceKind startSize else s

def longestMatchFnAux (left? : Option Syntax) (startSize startLhsPrec : Nat) (startPos : String.Pos) (prevPrio : Nat) (ps : List (Parse
  let rec parse (prevPrio : Nat) (ps : List (Parser × Nat)) :=
    match ps with
    | []    => fun _ s => longestMatchMkResult startSize s
    | p::ps => fun c s =>
      let (s, prevPrio) := longestMatchStep left? startSize startLhsPrec startPos prevPrio p.2 p.1.fn c s
      parse prevPrio ps c s
  parse prevPrio ps
```

```
def longestMatchFn (left? : Option Syntax) : List (Parser × Nat) → ParserFn
  | []    => fun _ s => s.mkError "longestMatch: empty list"
  | [p]   => fun c s => runLongestMatchParser left? s.lhsPrec p.1.fn c s
  | p::ps => fun c s =>
    let startSize := s.stackSize
    let startLhsPrec := s.lhsPrec
    let startPos  := s.pos
    let s         := runLongestMatchParser left? s.lhsPrec p.1.fn c s
    longestMatchFnAux left? startSize startLhsPrec startPos p.2 ps c s

def anyOfFn : List Parser → ParserFn
  | [],    _, s => s.mkError "anyOf: empty list"
  | [p],   c, s => p.fn c s
  | p::ps, c, s => orelseFn p.fn (anyOfFn ps) c s

@[inline] def checkColGeFn (errorMsg : String) : ParserFn := fun c s =>
  match c.savedPos? with
  | none => s
  | some savedPos =>
    let savedPos := c.fileMap.toPosition savedPos
    let pos      := c.fileMap.toPosition s.pos
    if pos.column ≥ savedPos.column then s
    else s.mkError errorMsg

@[inline] def checkColGe (errorMsg : String := "checkColGe") : Parser :=
  { fn := checkColGeFn errorMsg }

@[inline] def checkColGtFn (errorMsg : String) : ParserFn := fun c s =>
  match c.savedPos? with
  | none => s
  | some savedPos =>
    let savedPos := c.fileMap.toPosition savedPos
    let pos      := c.fileMap.toPosition s.pos
    if pos.column > savedPos.column then s
    else s.mkError errorMsg

@[inline] def checkColGt (errorMsg : String := "checkColGt") : Parser :=
  { fn := checkColGtFn errorMsg }

@[inline] def checkLineEqFn (errorMsg : String) : ParserFn := fun c s =>
  match c.savedPos? with
  | none => s
  | some savedPos =>
    let savedPos := c.fileMap.toPosition savedPos
    let pos      := c.fileMap.toPosition s.pos
    if pos.line == savedPos.line then s
    else s.mkError errorMsg
```

```
@[inline] def checkLineEq (errorMsg : String := "checkLineEq") : Parser :=
  { fn := checkLineEqFn errorMsg }

@[inline] def withPosition (p : Parser) : Parser := {
  info := p.info,
  fn   := fun c s =>
    p.fn { c with savedPos? := s.pos } s
}

@[inline] def withoutPosition (p : Parser) : Parser := {
  info := p.info,
  fn   := fun c s =>
    let pos := c.fileMap.toPosition s.pos
    p.fn { c with savedPos? := none } s
}

@[inline] def withForbidden (tk : Token) (p : Parser) : Parser := {
  info := p.info,
  fn   := fun c s => p.fn { c with forbiddenTk? := tk } s
}

@[inline] def withoutForbidden (p : Parser) : Parser := {
  info := p.info,
  fn   := fun c s => p.fn { c with forbiddenTk? := none } s
}

def eoiFn : ParserFn := fun c s =>
  let i := s.pos
  if c.input.atEnd i then s
  else s.mkError "expected end of file"

@[inline] def eoi : Parser :=
  { fn := eoiFn }

open Std (RBMap RBMap.empty)

/-- A multimap indexed by tokens. Used for indexing parsers by their leading token. -/
def TokenMap (α : Type) := RBMap Name (List α) Name.quickLt

namespace TokenMap

def insert (map : TokenMap α) (k : Name) (v : α) : TokenMap α :=
  match map.find? k with
  | none    => Std.RBMap.insert map k [v]
  | some vs => Std.RBMap.insert map k (v::vs)
```

```
instance : Inhabited (TokenMap α) := ⟨RBMap.empty⟩

instance : EmptyCollection (TokenMap α) := ⟨RBMap.empty⟩

end TokenMap

structure PrattParsingTables where
  leadingTable    : TokenMap (Parser × Nat) := {}
  leadingParsers  : List (Parser × Nat) := [] -- for supporting parsers we cannot obtain first token
  trailingTable   : TokenMap (Parser × Nat) := {}
  trailingParsers : List (Parser × Nat) := [] -- for supporting parsers such as function application

instance : Inhabited PrattParsingTables := ⟨{}⟩

/-
  The type `leadingIdentBehavior` specifies how the parsing table
  lookup function behaves for identifiers.  The function `prattParser`
  uses two tables `leadingTable` and `trailingTable`. They map tokens
  to parsers.

  - `LeadingIdentBehavior.default`: if the leading token
    is an identifier, then `prattParser` just executes the parsers
    associated with the auxiliary token "ident".

  - `LeadingIdentBehavior.symbol`: if the leading token is
    an identifier `<foo>`, and there are parsers `P` associated with
    the toek `<foo>`, then it executes `P`. Otherwise, it executes
    only the parsers associated with the auxiliary token "ident".

  - `LeadingIdentBehavior.both`: if the leading token
    an identifier `<foo>`, the it executes the parsers associated
    with token `<foo>` and parsers associated with the auxiliary
    token "ident".

  We use `LeadingIdentBehavior.symbol` and `LeadingIdentBehavior.both`
  and `nonReservedSymbol` parser to implement the `tactic` parsers.
  The idea is to avoid creating a reserved symbol for each
  builtin tactic (e.g., `apply`, `assumption`, etc.).  That is, users
  may still use these symbols as identifiers (e.g., naming a
  function).
-/

inductive LeadingIdentBehavior where
  | default
  | symbol
  | both
  deriving Inhabited, BEq
```

```
/--
  Each parser category is implemented using a Pratt's parser.
  The system comes equipped with the following categories: `level`, `term`, `tactic`, and `command`.
  Users and plugins may define extra categories.

  The method
  ```
  categoryParser `term prec
  ```
  executes the Pratt's parser for category `term` with precedence `prec`.
  That is, only parsers with precedence at least `prec` are considered.
  The method `termParser prec` is equivalent to the method above.
-/
structure ParserCategory where
  tables   : PrattParsingTables
  behavior : LeadingIdentBehavior
  deriving Inhabited

abbrev ParserCategories := Std.PersistentHashMap Name ParserCategory

def indexed {α : Type} (map : TokenMap α) (c : ParserContext) (s : ParserState) (behavior : LeadingIdentBehavior) : ParserState × List
  let (s, stx) := peekToken c s
  let find (n : Name) : ParserState × List α :=
    match map.find? n with
    | some as => (s, as)
    | _       => (s, [])
  match stx with
  | Except.ok (Syntax.atom _ sym)      => find (Name.mkSimple sym)
  | Except.ok (Syntax.ident _ _ val _) =>
    match behavior with
    | LeadingIdentBehavior.default => find identKind
    | LeadingIdentBehavior.symbol =>
      match map.find? val with
      | some as => (s, as)
      | none    => find identKind
    | LeadingIdentBehavior.both =>
      match map.find? val with
      | some as => match map.find? identKind with
        | some as' => (s, as ++ as')
        | _        => (s, as)
      | none    => find identKind
  | Except.ok (Syntax.node k _)        => find k
  | Except.ok _                        => (s, [])
  | Except.error s'                    => (s', [])
```

```
abbrev CategoryParserFn := Name → ParserFn

builtin_initialize categoryParserFnRef : IO.Ref CategoryParserFn ← IO.mkRef fun _ => whitespace

builtin_initialize categoryParserFnExtension : EnvExtension CategoryParserFn ← registerEnvExtension $ categoryParserFnRef.get

def categoryParserFn (catName : Name) : ParserFn := fun ctx s =>
  categoryParserFnExtension.getState ctx.env catName ctx s

def categoryParser (catName : Name) (prec : Nat) : Parser := {
  fn := fun c s => categoryParserFn catName { c with prec := prec } s
}

-- Define `termParser` here because we need it for antiquotations
@[inline] def termParser (prec : Nat := 0) : Parser :=
  categoryParser `term prec

/- ============== -/
/- Antiquotations -/
/- ============== -/

/-- Fail if previous token is immediately followed by ':'. -/
def checkNoImmediateColon : Parser := {
  fn := fun c s =>
    let prev := s.stxStack.back
    if checkTailNoWs prev then
      let input := c.input
      let i     := s.pos
      if input.atEnd i then s
      else
        let curr := input.get i
        if curr == ':' then
          s.mkUnexpectedError "unexpected ':'"
        else s
    else s
}

def setExpectedFn (expected : List String) (p : ParserFn) : ParserFn := fun c s =>
  match p c s with
    | s'@{ errorMsg := some msg, .. } => { s' with errorMsg := some { msg with expected := [] } }
    | s'                              => s'

def setExpected (expected : List String) (p : Parser) : Parser :=
  { fn := setExpectedFn expected p.fn, info := p.info }

def pushNone : Parser :=
  { fn := fun c s => s.pushSyntax mkNullNode }
```

```
-- We support two kinds of antiquotations: `$id` and `$(t)`, where `id` is a term identifier and `t` is a term.
def antiquotNestedExpr : Parser := node `antiquotNestedExpr (symbolNoAntiquot "(" >> toggleInsideQuot termParser >> symbolNoAntiquot ")")
def antiquotExpr : Parser       := identNoAntiquot <|> antiquotNestedExpr

@[inline] def tokenWithAntiquotFn (p : ParserFn) : ParserFn := fun c s => do
  let s := p c s
  if s.hasError then
    return s
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := (checkNoWsBefore >> symbolNoAntiquot "%" >> symbolNoAntiquot "$" >> checkNoWsBefore >> antiquotExpr).fn c s
  if s.hasError then
    return s.restore iniSz iniPos
  s.mkNode (`token_antiquot) (iniSz - 1)

@[inline] def tokenWithAntiquot (p : Parser) : Parser where
  fn   := tokenWithAntiquotFn p.fn
  info := p.info

@[inline] def symbol (sym : String) : Parser :=
  tokenWithAntiquot (symbolNoAntiquot sym)

instance : Coe String Parser := ⟨fun s => symbol s ⟩

@[inline] def nonReservedSymbol (sym : String) (includeIdent := false) : Parser :=
  tokenWithAntiquot (nonReservedSymbolNoAntiquot sym includeIdent)

@[inline] def unicodeSymbol (sym asciiSym : String) : Parser :=
  tokenWithAntiquot (unicodeSymbolNoAntiquot sym asciiSym)

/--
  Define parser for `$e` (if anonymous == true) and `$e:name`. Both
  forms can also be used with an appended `*` to turn them into an
  antiquotation "splice". If `kind` is given, it will additionally be checked
  when evaluating `match_syntax`. Antiquotations can be escaped as in `$$e`, which
  produces the syntax tree for `$e`. -/
def mkAntiquot (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Parser :=
  let kind := (kind.getD Name.anonymous) ++ `antiquot
  let nameP := node `antiquotName $ checkNoWsBefore ("no space before ':" ++ name ++ "'") >> symbol ":" >> nonReservedSymbol name
  -- if parsing the kind fails and `anonymous` is true, check that we're not ignoring a different
  -- antiquotation kind via `noImmediateColon`
  let nameP := if anonymous then nameP <|> checkNoImmediateColon >> pushNone else nameP
  -- antiquotations are not part of the "standard" syntax, so hide "expected '$'" on error
  leadingNode kind maxPrec $ atomic $
    setExpected [] "$" >>
    manyNoAntiquot (checkNoWsBefore "" >> "$") >>
```

```
    checkNoWsBefore "no space before spliced term" >> antiquotExpr >>
    nameP

def tryAnti (c : ParserContext) (s : ParserState) : Bool :=
  let (s, stx) := peekToken c s
  match stx with
  | Except.ok stx@(Syntax.atom _ sym) => sym == "$"
  | _                                  => false

@[inline] def withAntiquotFn (antiquotP p : ParserFn) : ParserFn := fun c s =>
  if tryAnti c s then orelseFn antiquotP p c s else p c s

/-- Optimized version of `mkAntiquot ... <|> p`. -/
@[inline] def withAntiquot (antiquotP p : Parser) : Parser := {
  fn := withAntiquotFn antiquotP.fn p.fn,
  info := orelseInfo antiquotP.info p.info
}

def withoutInfo (p : Parser) : Parser :=
  { fn := p.fn }

/-- Parse `$[p]suffix`, e.g. `$[p],*`. -/
def mkAntiquotSplice (kind : SyntaxNodeKind) (p suffix : Parser) : Parser :=
  let kind := kind ++ `antiquot_scope
  leadingNode kind maxPrec $ atomic $
    setExpected [] "$" >>
    manyNoAntiquot (checkNoWsBefore "" >> "$") >>
    checkNoWsBefore "no space before spliced term" >> symbol "[" >> node nullKind p >> symbol "]" >>
    suffix

@[inline] def withAntiquotSuffixSpliceFn (kind : SyntaxNodeKind) (p suffix : ParserFn) : ParserFn := fun c s => do
  let s := p c s
  if s.hasError || !s.stxStack.back.isAntiquot then
    return s
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s      := suffix c s
  if s.hasError then
    return s.restore iniSz iniPos
  s.mkNode (kind ++ `antiquot_suffix_splice) (s.stxStack.size - 2)

/-- Parse `suffix` after an antiquotation, e.g. `$x,*`, and put both into a new node. -/
@[inline] def withAntiquotSuffixSplice (kind : SyntaxNodeKind) (p suffix : Parser) : Parser := {
  info := andthenInfo p.info suffix.info,
  fn   := withAntiquotSuffixSpliceFn kind p.fn suffix.fn
}
```

```
def withAntiquotSpliceAndSuffix (kind : SyntaxNodeKind) (p suffix : Parser) :=
  -- prevent `p`'s info from being collected twice
  withAntiquot (mkAntiquotSplice kind (withoutInfo p) suffix) (withAntiquotSuffixSplice kind p suffix)

def nodeWithAntiquot (name : String) (kind : SyntaxNodeKind) (p : Parser) (anonymous := false) : Parser :=
  withAntiquot (mkAntiquot name kind anonymous) $ node kind p

/- ==================== -/
/- End of Antiquotations -/
/- ==================== -/

def sepByElemParser (p : Parser) (sep : String) : Parser :=
  withAntiquotSpliceAndSuffix `sepBy p (symbol (sep.trim ++ "*"))

def sepBy (p : Parser) (sep : String) (psep : Parser := symbol sep) (allowTrailingSep : Bool := false) : Parser :=
  sepByNoAntiquot (sepByElemParser p sep) psep allowTrailingSep

def sepBy1 (p : Parser) (sep : String) (psep : Parser := symbol sep) (allowTrailingSep : Bool := false) : Parser :=
  sepBy1NoAntiquot (sepByElemParser p sep) psep allowTrailingSep

def categoryParserOfStackFn (offset : Nat) : ParserFn := fun ctx s =>
  let stack := s.stxStack
  if stack.size < offset + 1 then
    s.mkUnexpectedError ("failed to determine parser category using syntax stack, stack is too small")
  else
    match stack.get! (stack.size - offset - 1) with
    | Syntax.ident _ _ catName _ => categoryParserFn catName ctx s
    | _ => s.mkUnexpectedError ("failed to determine parser category using syntax stack, the specified element on the stack is not an i

def categoryParserOfStack (offset : Nat) (prec : Nat := 0) : Parser :=
  { fn := fun c s => categoryParserOfStackFn offset { c with prec := prec } s }

unsafe def evalParserConstUnsafe (declName : Name) : ParserFn := fun ctx s =>
  match ctx.env.evalConstCheck Parser ctx.options `Lean.Parser.Parser declName <|>
        ctx.env.evalConstCheck Parser ctx.options `Lean.Parser.TrailingParser declName with
  | Except.ok p    => p.fn ctx s
  | Except.error e => s.mkUnexpectedError s!"error running parser {declName}: {e}"

@[implementedBy evalParserConstUnsafe]
constant evalParserConst (declName : Name) : ParserFn

unsafe def parserOfStackFnUnsafe (offset : Nat) : ParserFn := fun ctx s =>
  let stack := s.stxStack
  if stack.size < offset + 1 then
    s.mkUnexpectedError ("failed to determine parser using syntax stack, stack is too small")
  else
    match stack.get! (stack.size - offset - 1) with
```

```
    | Syntax.ident (val := parserName) .. =>
      match ctx.resolveName parserName with
      | [(parserName, [])] =>
        let iniSz := s.stackSize
        let s := evalParserConst parserName ctx s
        if !s.hasError && s.stackSize != iniSz + 1 then
          s.mkUnexpectedError "expected parser to return exactly one syntax object"
        else
          s
      | _::_::_ => s.mkUnexpectedError s!"ambiguous parser name {parserName}"
      | _ => s.mkUnexpectedError s!"unknown parser {parserName}"
    | _ => s.mkUnexpectedError ("failed to determine parser using syntax stack, the specified element on the stack is not an identifier

@[implementedBy parserOfStackFnUnsafe]
constant parserOfStackFn (offset : Nat) : ParserFn

def parserOfStack (offset : Nat) (prec : Nat := 0) : Parser :=
  { fn := fun c s => parserOfStackFn offset { c with prec := prec } s }

/-- Run `declName` if possible and inside a quotation, or else `p`. The `ParserInfo` will always be taken from `p`. -/
def evalInsideQuot (declName : Name) (p : Parser) : Parser := { p with
  fn := fun c s =>
    if c.insideQuot && c.env.contains declName then
      evalParserConst declName c s
    else
      p.fn c s }

private def mkResult (s : ParserState) (iniSz : Nat) : ParserState :=
  if s.stackSize == iniSz + 1 then s
  else s.mkNode nullKind iniSz -- throw error instead?

def leadingParserAux (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) : ParserFn := fun c s => do
  let iniSz   := s.stackSize
  let (s, ps) := indexed tables.leadingTable c s behavior
  if s.hasError then
    return s
  let ps      := tables.leadingParsers ++ ps
  if ps.isEmpty then
    return s.mkError (toString kind)
  let s := longestMatchFn none ps c s
  mkResult s iniSz

@[inline] def leadingParser (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) (antiquotParser : ParserFn) :
  withAntiquotFn antiquotParser (leadingParserAux kind tables behavior)

def trailingLoopStep (tables : PrattParsingTables) (left : Syntax) (ps : List (Parser × Nat)) : ParserFn := fun c s =>
  longestMatchFn left (ps ++ tables.trailingParsers) c s
```

```
partial def trailingLoop (tables : PrattParsingTables) (c : ParserContext) (s : ParserState) : ParserState := do
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let (s, ps)       := indexed tables.trailingTable c s LeadingIdentBehavior.default
  if s.hasError then
    -- Discard token parse errors and break the trailing loop instead.
    -- The error will be flagged when the next leading position is parsed, unless the token
    -- is in fact valid there (e.g. EOI at command level, no-longer forbidden token)
    return s.restore iniSz iniPos
  if ps.isEmpty && tables.trailingParsers.isEmpty then
    return s -- no available trailing parser
  let left   := s.stxStack.back
  let s      := s.popSyntax
  let s      := trailingLoopStep tables left ps c s
  if s.hasError then
    -- Discard non-consuming parse errors and break the trailing loop instead, restoring `left`.
    -- This is necessary for fallback parsers like `app` that pretend to be always applicable.
    return if s.pos == iniPos then s.restore (iniSz - 1) iniPos |>.pushSyntax left else s
  trailingLoop tables c s

/--

  Implements a variant of Pratt's algorithm. In Pratt's algorithms tokens have a right and left binding power.
  In our implementation, parsers have precedence instead. This method selects a parser (or more, via
  `longestMatchFn`) from `leadingTable` based on the current token. Note that the unindexed `leadingParsers` parsers
  are also tried. We have the unidexed `leadingParsers` because some parsers do not have a "first token". Example:
  ```
  syntax term:51 "≤" ident "<" term "|" term : index
  ```
  Example, in principle, the set of first tokens for this parser is any token that can start a term, but this set
  is always changing. Thus, this parsing rule is stored as an unindexed leading parser at `leadingParsers`.
  After processing the leading parser, we chain with parsers from `trailingTable`/`trailingParsers` that have precedence
  at least `c.prec` where `c` is the `ParsingContext`. Recall that `c.prec` is set by `categoryParser`.

  Note that in the original Pratt's algorith, precedences are only checked before calling trailing parsers. In our
  implementation, leading *and* trailing parsers check the precendece. We claim our algorithm is more flexible,
  modular and easier to understand.

  `antiquotParser` should be a `mkAntiquot` parser (or always fail) and is tried before all other parsers.
  It should not be added to the regular leading parsers because it would heavily
  overlap with antiquotation parsers nested inside them. -/
@[inline] def prattParser (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) (antiquotParser : ParserFn) : P
  let iniSz  := s.stackSize
  let iniPos := s.pos
  let s := leadingParser kind tables behavior antiquotParser c s
  if s.hasError then
```

```
    s
  else
    trailingLoop tables c s

def fieldIdxFn : ParserFn := fun c s =>
  let initStackSz := s.stackSize
  let iniPos := s.pos
  let curr     := c.input.get iniPos
  if curr.isDigit && curr != '0' then
    let s     := takeWhileFn (fun c => c.isDigit) c s
    mkNodeToken fieldIdxKind iniPos c s
  else
    s.mkErrorAt "field index" iniPos initStackSz

@[inline] def fieldIdx : Parser :=
  withAntiquot (mkAntiquot "fieldIdx" `fieldIdx) {
    fn   := fieldIdxFn,
    info := mkAtomicInfo "fieldIdx"
  }

@[inline] def skip : Parser := {
  fn   := fun c s => s,
  info := epsilonInfo
}

end Parser

namespace Syntax

section
variable {β : Type} {m : Type → Type} [Monad m]

@[inline] def foldArgsM (s : Syntax) (f : Syntax → β → m β) (b : β) : m β :=
  s.getArgs.foldlM (flip f) b

@[inline] def foldArgs (s : Syntax) (f : Syntax → β → β) (b : β) : β :=
  Id.run (s.foldArgsM f b)

@[inline] def forArgsM (s : Syntax) (f : Syntax → m Unit) : m Unit :=
  s.foldArgsM (fun s _ => f s) ()
end

end Syntax
end Lean
::::::::::::::
Parser/Command.lean
::::::::::::::
```

```
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Term
import Lean.Parser.Do

namespace Lean
namespace Parser

/--
  Syntax quotation for terms and (lists of) commands. We prefer terms, so ambiguous quotations like
  `($x $y) will be parsed as an application, not two commands. Use `($x:command $y:command) instead.
  Multiple command will be put in a `null node, but a single command will not (so that you can directly
  match against a quotation in a command kind's elaborator). -/
-- TODO: use two separate quotation parsers with parser priorities instead
@[builtinTermParser] def Term.quot := leading_parser "`(" >> toggleInsideQuot (termParser <|> many1Unbox commandParser) >> ")"

namespace Command

def namedPrio := leading_parser (atomic ("(" >> nonReservedSymbol "priority") >> " := " >> priorityParser >> ")")
def optNamedPrio := optional namedPrio

def «private»        := leading_parser "private "
def «protected»      := leading_parser "protected "
def visibility       := «private» <|> «protected»
def «noncomputable»  := leading_parser "noncomputable "
def «unsafe»         := leading_parser "unsafe "
def «partial»        := leading_parser "partial "
def declModifiers (inline : Bool) := leading_parser optional docComment >> optional (Term.«attributes» >> if inline then skip else ppDe
def declId           := leading_parser ident >> optional (".{" >> sepBy1 ident ", " >> "}")
def declSig          := leading_parser many (ppSpace >> (Term.simpleBinderWithoutType <|> Term.bracketedBinder)) >> Term.typeSpec
def optDeclSig       := leading_parser many (ppSpace >> (Term.simpleBinderWithoutType <|> Term.bracketedBinder)) >> Term.optType
def declValSimple    := leading_parser " :=\n" >> termParser >> optional Term.whereDecls
def declValEqns      := leading_parser Term.matchAltsWhereDecls
def declVal          := declValSimple <|> declValEqns <|> Term.whereDecls
def «abbrev»         := leading_parser "abbrev " >> declId >> optDeclSig >> declVal
def «def»            := leading_parser "def " >> declId >> optDeclSig >> declVal
def «theorem»        := leading_parser "theorem " >> declId >> declSig >> declVal
def «constant»       := leading_parser "constant " >> declId >> declSig >> optional declValSimple
def «instance»       := leading_parser Term.attrKind >> "instance " >> optNamedPrio >> optional declId >> declSig >> declVal
def «axiom»          := leading_parser "axiom " >> declId >> declSig
def «example»        := leading_parser "example " >> declSig >> declVal
def inferMod         := leading_parser atomic (symbol "{" >> "}")
def ctor             := leading_parser "\n| " >> declModifiers true >> ident >> optional inferMod >> optDeclSig
def optDeriving      := leading_parser optional (atomic ("deriving " >> notSymbol "instance") >> sepBy1 ident ", ")
```

```
def «inductive»      := leading_parser "inductive " >> declId >> optDeclSig >> optional (symbol ":=" <|> "where") >> many ctor >> optDe
def classInductive   := leading_parser atomic (group (symbol "class " >> "inductive ")) >> declId >> optDeclSig >> optional (symbol ":=
def structExplicitBinder := leading_parser atomic (declModifiers true >> "(") >> many1 ident >> optional inferMod >> optDeclSig >> opti
def structImplicitBinder := leading_parser atomic (declModifiers true >> "{") >> many1 ident >> optional inferMod >> declSig >> "}"
def structInstBinder     := leading_parser atomic (declModifiers true >> "[") >> many1 ident >> optional inferMod >> declSig >> "]"
def structSimpleBinder   := leading_parser atomic (declModifiers true >> ident) >> optional inferMod >> optDeclSig >> optional Term.bin
def structFields         := leading_parser manyIndent (ppLine >> checkColGe >>(structExplicitBinder <|> structImplicitBinder <|> struct
def structCtor           := leading_parser atomic (declModifiers true >> ident >> optional inferMod >> " :: ")
def structureTk          := leading_parser "structure "
def classTk              := leading_parser "class "
def «extends»            := leading_parser " extends " >> sepBy1 termParser ", "
def «structure»          := leading_parser
    (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType
    >> optional ((symbol " := " <|> " where ") >> optional structCtor >> structFields)
    >> optDeriving
@[builtinCommandParser] def declaration := leading_parser
declModifiers false >> («abbrev» <|> «def» <|> «theorem» <|> «constant» <|> «instance» <|> «axiom» <|> «example» <|> «inductive» <|> cl
@[builtinCommandParser] def «deriving»     := leading_parser "deriving " >> "instance " >> sepBy1 ident ", " >> " for " >> sepBy1 ident
@[builtinCommandParser] def «section»      := leading_parser "section " >> optional ident
@[builtinCommandParser] def «namespace»    := leading_parser "namespace " >> ident
@[builtinCommandParser] def «end»          := leading_parser "end " >> optional ident
@[builtinCommandParser] def «variable»     := leading_parser "variable" >> many1 Term.bracketedBinder
@[builtinCommandParser] def «universe»     := leading_parser "universe " >> ident
@[builtinCommandParser] def «universes»    := leading_parser "universes " >> many1 ident
@[builtinCommandParser] def check          := leading_parser "#check " >> termParser
@[builtinCommandParser] def check_failure  := leading_parser "#check_failure " >> termParser -- Like `#check`, but succeeds only if ter
@[builtinCommandParser] def reduce         := leading_parser "#reduce " >> termParser
@[builtinCommandParser] def eval           := leading_parser "#eval " >> termParser
@[builtinCommandParser] def synth          := leading_parser "#synth " >> termParser
@[builtinCommandParser] def exit           := leading_parser "#exit"
@[builtinCommandParser] def print          := leading_parser "#print " >> (ident <|> strLit)
@[builtinCommandParser] def printAxioms    := leading_parser "#print " >> nonReservedSymbol "axioms " >> ident
@[builtinCommandParser] def «resolve_name» := leading_parser "#resolve_name " >> ident
@[builtinCommandParser] def «init_quot»    := leading_parser "init_quot"
def optionValue := nonReservedSymbol "true" <|> nonReservedSymbol "false" <|> strLit <|> numLit
@[builtinCommandParser] def «set_option»   := leading_parser "set_option " >> ident >> ppSpace >> optionValue
def eraseAttr := leading_parser "-" >> ident
@[builtinCommandParser] def «attribute»    := leading_parser "attribute " >> "[" >> sepBy1 (eraseAttr <|> Term.attrInstance) ", " >> "]
@[builtinCommandParser] def «export»       := leading_parser "export " >> ident >> "(" >> many1 ident >> ")"
def openHiding       := leading_parser atomic (ident >> "hiding") >> many1 ident
def openRenamingItem := leading_parser ident >> unicodeSymbol "→" "->" >> ident
def openRenaming     := leading_parser atomic (ident >> "renaming") >> sepBy1 openRenamingItem ", "
def openOnly         := leading_parser atomic (ident >> "(") >> many1 ident >> ")"
def openSimple       := leading_parser many1 ident
def openDecl         := openHiding <|> openRenaming <|> openOnly <|> openSimple
@[builtinCommandParser] def «open»    := leading_parser "open " >> openDecl
```

```
@[builtinCommandParser] def «mutual» := leading_parser "mutual " >> many1 (ppLine >> notSymbol "end" >> commandParser) >> ppDedent (ppL
@[builtinCommandParser] def «initialize» := leading_parser "initialize " >> optional (atomic (ident >> Term.typeSpec >> Term.leftArrow)
@[builtinCommandParser] def «builtin_initialize» := leading_parser "builtin_initialize " >> optional (atomic (ident >> Term.typeSpec >>

@[builtinCommandParser] def «in»  := trailing_parser " in " >> commandParser

@[runBuiltinParserAttributeHooks] abbrev declModifiersF := declModifiers false
@[runBuiltinParserAttributeHooks] abbrev declModifiersT := declModifiers true

builtin_initialize
  register_parser_alias "declModifiers"       declModifiersF
  register_parser_alias "nestedDeclModifiers" declModifiersT
  register_parser_alias "declId"              declId
  register_parser_alias "declSig"             declSig
  register_parser_alias "declVal"             declVal
  register_parser_alias "optDeclSig"          optDeclSig
  register_parser_alias "openDecl"            openDecl

end Command

namespace Term
@[builtinTermParser] def «open» := leading_parser:leadPrec "open " >> Command.openDecl >> " in " >> termParser
@[builtinTermParser] def «set_option» := leading_parser:leadPrec "set_option " >> ident >> ppSpace >> Command.optionValue >> " in " >>
end Term

namespace Tactic
@[builtinTacticParser] def «open» := leading_parser:leadPrec "open " >> Command.openDecl >> " in " >> tacticSeq
@[builtinTacticParser] def «set_option» := leading_parser:leadPrec "set_option " >> ident >> ppSpace >> Command.optionValue >> " in " >
end Tactic

end Parser
end Lean
::::::::::::::
Parser/Do.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Term

namespace Lean
namespace Parser

builtin_initialize registerBuiltinParserAttribute `builtinDoElemParser `doElem
builtin_initialize registerBuiltinDynamicParserAttribute `doElemParser `doElem
```

```
@[inline] def doElemParser (rbp : Nat := 0) : Parser :=
  categoryParser `doElem rbp

namespace Term
def leftArrow : Parser := unicodeSymbol " ← " " <- "
@[builtinTermParser] def liftMethod := leading_parser:minPrec leftArrow >> termParser

def doSeqItem      := leading_parser ppLine >> doElemParser >> optional "; "
def doSeqIndent    := leading_parser many1Indent doSeqItem
def doSeqBracketed := leading_parser "{" >> withoutPosition (many1 doSeqItem) >> ppLine >> "}"
def doSeq          := doSeqBracketed <|> doSeqIndent

def termBeforeDo := withForbidden "do" termParser

attribute [runBuiltinParserAttributeHooks] doSeq termBeforeDo

builtin_initialize
  register_parser_alias "doSeq" doSeq
  register_parser_alias "termBeforeDo" termBeforeDo

def notFollowedByRedefinedTermToken :=
  -- Remark: we don't currently support `open` and `set_option` in `do`-blocks, but we include them in the following list to fix the am
  -- "open" command following `do`-block. If we don't add `do`, then users would have to indent `do` blocks or use `{ ... }`.
  notFollowedBy ("set_option" <|> "open" <|> "if" <|> "match" <|> "let" <|> "have" <|> "do" <|> "dbg_trace" <|> "assert!" <|> "for" <|>

@[builtinDoElemParser] def doLet      := leading_parser "let " >> optional "mut " >> letDecl

@[builtinDoElemParser] def doLetRec   := leading_parser group ("let " >> nonReservedSymbol "rec ") >> letRecDecls
def doIdDecl   := leading_parser atomic (ident >> optType >> leftArrow) >> doElemParser
def doPatDecl  := leading_parser atomic (termParser >> leftArrow) >> doElemParser >> optional (checkColGt >> " | " >> doElemParser)
@[builtinDoElemParser] def doLetArrow      := leading_parser withPosition ("let " >> optional "mut " >> (doIdDecl <|> doPatDecl))

-- We use `letIdDeclNoBinders` to define `doReassign`.
-- Motivation: we do not reassign functions, and avoid parser conflict
def letIdDeclNoBinders := node `Lean.Parser.Term.letIdDecl $ atomic (ident >> pushNone >> optType >> " := ") >> termParser

@[builtinDoElemParser] def doReassign      := leading_parser notFollowedByRedefinedTermToken >> (letIdDeclNoBinders <|> letPatDecl)
@[builtinDoElemParser] def doReassignArrow := leading_parser notFollowedByRedefinedTermToken >> withPosition (doIdDecl <|> doPatDecl)
@[builtinDoElemParser] def doHave     := leading_parser "have " >> Term.haveDecl
/-
In `do` blocks, we support `if` without an `else`. Thus, we use indentation to prevent examples such as
```
if c_1 then
  if c_2 then
    action_1
else
```
```

```
  action_2
```
from being parsed as
```
if c_1 then {
  if c_2 then {
    action_1
  } else {
    action_2
  }
}
```
We also have special support for `else if` because we don't want to write
```
if c_1 then
  action_1
else if c_2 then
      action_2
    else
      action_3
```
-/
def elseIf := atomic (group (withPosition (" else " >> checkLineEq >> " if ")))
-- ensure `if $e then ...` still binds to `e:term`
def doIfLetPure := leading_parser " := " >> termParser
def doIfLetBind := leading_parser " ← " >> termParser
def doIfLet   := nodeWithAntiquot "doIfLet"   `Lean.Parser.Term.doIfLet   <| "let " >> termParser >> (doIfLetPure <|> doIfLetBi
def doIfProp   := nodeWithAntiquot "doIfProp"   `Lean.Parser.Term.doIfProp   <| optIdent >> termParser
def doIfCond   := withAntiquot (mkAntiquot "doIfCond" none (anonymous := false)) <| doIfLet <|> doIfProp
@[builtinDoElemParser] def doIf := leading_parser withPosition $
  "if " >> doIfCond >> " then " >> doSeq
  >> many (checkColGe "'else if' in 'do' must be indented" >> group (elseIf >> doIfCond >> " then " >> doSeq))
  >> optional (checkColGe "'else' in 'do' must be indented" >> " else " >> doSeq)
@[builtinDoElemParser] def doUnless := leading_parser "unless " >> withForbidden "do" termParser >> "do " >> doSeq
def doForDecl := leading_parser termParser >> " in " >> withForbidden "do" termParser
@[builtinDoElemParser] def doFor   := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq

def doMatchAlts := matchAlts (rhsParser := doSeq)
@[builtinDoElemParser] def doMatch := leading_parser:leadPrec "match " >> optional Term.generalizingParam >> sepBy1 matchDiscr ", " >>

def doCatch   := leading_parser atomic ("catch " >> binderIdent) >> optional (" : " >> termParser) >> darrow >> doSeq
def doCatchMatch := leading_parser "catch " >> doMatchAlts
def doFinally   := leading_parser "finally " >> doSeq
@[builtinDoElemParser] def doTry   := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally

@[builtinDoElemParser] def doBreak   := leading_parser "break"
@[builtinDoElemParser] def doContinue := leading_parser "continue"
```
```

```
@[builtinDoElemParser] def doReturn    := leading_parser:leadPrec withPosition ("return " >> optional (checkLineEq >> termParser))
@[builtinDoElemParser] def doDbgTrace  := leading_parser:leadPrec "dbg_trace " >> ((interpolatedStr termParser) <|> termParser)
@[builtinDoElemParser] def doAssert    := leading_parser:leadPrec "assert! " >> termParser

/-
We use `notFollowedBy` to avoid counterintuitive behavior.

For example, the `if`-term parser
doesn't enforce indentation restrictions, but we don't want it to be used when `doIf` fails.
Note that parser priorities would not solve this problem since the `doIf` parser is failing while the `if`
parser is succeeding. The first `notFollowedBy` prevents this problem.

Consider the `doElem` `x := (a, b)` it contains an error since we are using `)` instead of `)`. Thus, `doReassign` parser fails.
However, `doExpr` would succeed consuming just `x`, and cryptic error message is generated after that.
The second `notFollowedBy` prevents this problem.
-/
@[builtinDoElemParser] def doExpr   := leading_parser notFollowedByRedefinedTermToken >> termParser >> notFollowedBy (symbol ":=" <|> s
@[builtinDoElemParser] def doNested := leading_parser "do " >> doSeq

@[builtinTermParser] def «do»  := leading_parser:argPrec "do " >> doSeq

@[builtinTermParser] def doElem.quot : Parser := leading_parser "`(doElem|" >> toggleInsideQuot doElemParser >> ")"

/- macros for using `unless`, `for`, `try`, `return` as terms. They expand into `do unless ...`, `do for ...`, `do try ...`, and `do re
@[builtinTermParser] def termUnless := leading_parser "unless " >> withForbidden "do" termParser >> "do " >> doSeq
@[builtinTermParser] def termFor    := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq
@[builtinTermParser] def termTry    := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally
@[builtinTermParser] def termReturn := leading_parser:leadPrec withPosition ("return " >> optional (checkLineEq >> termParser))

end Term
end Parser
end Lean
::::::::::::::
Parser/Extension.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.ScopedEnvExtension
import Lean.Parser.Basic
import Lean.Parser.StrInterpolation
import Lean.KeyedDeclsAttribute

/-! Extensible parsing via attributes -/

```

```
namespace Lean
namespace Parser

builtin_initialize builtinTokenTable : IO.Ref TokenTable ← IO.mkRef {}

/- Global table with all SyntaxNodeKind's -/
builtin_initialize builtinSyntaxNodeKindSetRef : IO.Ref SyntaxNodeKindSet ← IO.mkRef {}

def registerBuiltinNodeKind (k : SyntaxNodeKind) : IO Unit :=
  builtinSyntaxNodeKindSetRef.modify fun s => s.insert k

builtin_initialize
  registerBuiltinNodeKind choiceKind
  registerBuiltinNodeKind identKind
  registerBuiltinNodeKind strLitKind
  registerBuiltinNodeKind numLitKind
  registerBuiltinNodeKind scientificLitKind
  registerBuiltinNodeKind charLitKind
  registerBuiltinNodeKind nameLitKind

builtin_initialize builtinParserCategoriesRef : IO.Ref ParserCategories ← IO.mkRef {}

private def throwParserCategoryAlreadyDefined {α} (catName : Name) : ExceptT String Id α :=
  throw s!"parser category '{catName}' has already been defined"

private def addParserCategoryCore (categories : ParserCategories) (catName : Name) (initial : ParserCategory) : Except String ParserCat
  if categories.contains catName then
    throwParserCategoryAlreadyDefined catName
  else
    pure $ categories.insert catName initial

/-- All builtin parser categories are Pratt's parsers -/

private def addBuiltinParserCategory (catName : Name) (behavior : LeadingIdentBehavior) : IO Unit := do
  let categories ← builtinParserCategoriesRef.get
  let categories ← IO.ofExcept $ addParserCategoryCore categories catName { tables := {}, behavior := behavior}
  builtinParserCategoriesRef.set categories

namespace ParserExtension

inductive OLeanEntry where
  | token     (val : Token) : OLeanEntry
  | kind      (val : SyntaxNodeKind) : OLeanEntry
  | category  (catName : Name) (behavior : LeadingIdentBehavior)
  | parser    (catName : Name) (declName : Name) (prio : Nat) : OLeanEntry
  deriving Inhabited

```

```
inductive Entry where
  | token     (val : Token) : Entry
  | kind      (val : SyntaxNodeKind) : Entry
  | category  (catName : Name) (behavior : LeadingIdentBehavior)
  | parser    (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : Entry
  deriving Inhabited

def Entry.toOLeanEntry : Entry → OLeanEntry
  | token v             => OLeanEntry.token v
  | kind v              => OLeanEntry.kind v
  | category c b        => OLeanEntry.category c b
  | parser c d _ _ prio => OLeanEntry.parser c d prio

structure State where
  tokens      : TokenTable := {}
  kinds       : SyntaxNodeKindSet := {}
  categories  : ParserCategories := {}
  deriving Inhabited

end ParserExtension

open ParserExtension in
abbrev ParserExtension := ScopedEnvExtension OLeanEntry Entry State

private def ParserExtension.mkInitial : IO ParserExtension.State := do
  let tokens     ← builtinTokenTable.get
  let kinds      ← builtinSyntaxNodeKindSetRef.get
  let categories ← builtinParserCategoriesRef.get
  pure { tokens := tokens, kinds := kinds, categories := categories }

private def addTokenConfig (tokens : TokenTable) (tk : Token) : Except String TokenTable := do
  if tk == "" then throw "invalid empty symbol"
  else match tokens.find? tk with
    | none   => pure $ tokens.insert tk tk
    | some _ => pure tokens

def throwUnknownParserCategory {α} (catName : Name) : ExceptT String Id α :=
  throw s!"unknown parser category '{catName}'"

abbrev getCategory (categories : ParserCategories) (catName : Name) : Option ParserCategory :=
  categories.find? catName

def addLeadingParser (categories : ParserCategories) (catName : Name) (parserName : Name) (p : Parser) (prio : Nat) : Except String Par
  match getCategory categories catName with
  | none     =>
    throwUnknownParserCategory catName
  | some cat =>
```

```
    let addTokens (tks : List Token) : Except String ParserCategories :=
      let tks    := tks.map $ fun tk => Name.mkSimple tk
      let tables := tks.eraseDups.foldl (fun (tables : PrattParsingTables) tk => { tables with leadingTable := tables.leadingTable.inse
      pure $ categories.insert catName { cat with tables := tables }
    match p.info.firstTokens with
    | FirstTokens.tokens tks    => addTokens tks
    | FirstTokens.optTokens tks => addTokens tks
    | _ =>
      let tables := { cat.tables with leadingParsers := (p, prio) :: cat.tables.leadingParsers }
      pure $ categories.insert catName { cat with tables := tables }

private def addTrailingParserAux (tables : PrattParsingTables) (p : TrailingParser) (prio : Nat) : PrattParsingTables :=
  let addTokens (tks : List Token) : PrattParsingTables :=
    let tks := tks.map fun tk => Name.mkSimple tk
    tks.eraseDups.foldl (fun (tables : PrattParsingTables) tk => { tables with trailingTable := tables.trailingTable.insert tk (p, prio
  match p.info.firstTokens with
  | FirstTokens.tokens tks    => addTokens tks
  | FirstTokens.optTokens tks => addTokens tks
  | _                         => { tables with trailingParsers := (p, prio) :: tables.trailingParsers }

def addTrailingParser (categories : ParserCategories) (catName : Name) (p : TrailingParser) (prio : Nat) : Except String ParserCategori
  match getCategory categories catName with
  | none     => throwUnknownParserCategory catName
  | some cat => pure $ categories.insert catName { cat with tables := addTrailingParserAux cat.tables p prio }

def addParser (categories : ParserCategories) (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : Except St
  match leading, p with
  | true, p  => addLeadingParser categories catName declName p prio
  | false, p => addTrailingParser categories catName p prio

def addParserTokens (tokenTable : TokenTable) (info : ParserInfo) : Except String TokenTable :=
  let newTokens := info.collectTokens []
  newTokens.foldlM addTokenConfig tokenTable

private def updateBuiltinTokens (info : ParserInfo) (declName : Name) : IO Unit := do
  let tokenTable ← builtinTokenTable.swap {}
  match addParserTokens tokenTable info with
  | Except.ok tokenTable => builtinTokenTable.set tokenTable
  | Except.error msg     => throw (IO.userError s!"invalid builtin parser '{declName}', {msg}")

def addBuiltinParser (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : IO Unit := do
  let p := evalInsideQuot declName p
  let categories ← builtinParserCategoriesRef.get
  let categories ← IO.ofExcept $ addParser categories catName declName leading p prio
  builtinParserCategoriesRef.set categories
  builtinSyntaxNodeKindSetRef.modify p.info.collectKinds
  updateBuiltinTokens p.info declName
```

```
def addBuiltinLeadingParser (catName : Name) (declName : Name) (p : Parser) (prio : Nat) : IO Unit :=
  addBuiltinParser catName declName true p prio

def addBuiltinTrailingParser (catName : Name) (declName : Name) (p : TrailingParser) (prio : Nat) : IO Unit :=
  addBuiltinParser catName declName false p prio

def ParserExtension.addEntryImpl (s : State) (e : Entry) : State :=
  match e with
  | Entry.token tk =>
    match addTokenConfig s.tokens tk with
    | Except.ok tokens => { s with tokens := tokens }
    | _                => unreachable!
  | Entry.kind k =>
    { s with kinds := s.kinds.insert k }
  | Entry.category catName behavior =>
    if s.categories.contains catName then s
    else { s with
           categories := s.categories.insert catName { tables := {}, behavior := behavior } }
  | Entry.parser catName declName leading parser prio =>
    match addParser s.categories catName declName leading parser prio with
    | Except.ok categories => { s with categories := categories }
    | _ => unreachable!

unsafe def mkParserOfConstantUnsafe
    (categories : ParserCategories) (constName : Name) (compileParserDescr : ParserDescr → ImportM Parser) : ImportM (Bool × Parser) :=
  let env  := (← read).env
  let opts := (← read).opts
  match env.find? constName with
  | none      => throw ↑s!"unknow constant '{constName}'"
  | some info =>
    match info.type with
    | Expr.const `Lean.Parser.TrailingParser _ _ =>
      let p ← IO.ofExcept $ env.evalConst Parser opts constName
      pure ⟨false, p⟩
    | Expr.const `Lean.Parser.Parser _ _ =>
      let p ← IO.ofExcept $ env.evalConst Parser opts constName
      pure ⟨true, p⟩
    | Expr.const `Lean.ParserDescr _ _ =>
      let d ← IO.ofExcept $ env.evalConst ParserDescr opts constName
      let p ← compileParserDescr d
      pure ⟨true, p⟩
    | Expr.const `Lean.TrailingParserDescr _ _ =>
      let d ← IO.ofExcept $ env.evalConst TrailingParserDescr opts constName
      let p ← compileParserDescr d
      pure ⟨false, p⟩
    | _ => throw ↑s!"unexpected parser type at '{constName}' (`ParserDescr`, `TrailingParserDescr`, `Parser` or `TrailingParser` expect
```

```

@[implementedBy mkParserOfConstantUnsafe]
constant mkParserOfConstantAux
    (categories : ParserCategories) (constName : Name) (compileParserDescr : ParserDescr → ImportM Parser) : ImportM (Bool × Parser)

/- Parser aliases for making `ParserDescr` extensible -/
inductive AliasValue (α : Type) where
  | const  (p : α)
  | unary  (p : α → α)
  | binary (p : α → α → α)

abbrev AliasTable (α) := NameMap (AliasValue α)

def registerAliasCore {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) (value : AliasValue α) : IO Unit := do
  unless (← IO.initializing) do throw ↑"aliases can only be registered during initialization"
  if (← mapRef.get).contains aliasName then
    throw ↑s!"alias '{aliasName}' has already been declared"
  mapRef.modify (·.insert aliasName value)

def getAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (Option (AliasValue α)) := do
  return (← mapRef.get).find? aliasName

def getConstAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO α := do
  match (← getAlias mapRef aliasName) with
  | some (AliasValue.const v)  => pure v
  | some (AliasValue.unary _)  => throw ↑s!"parser '{aliasName}' is not a constant, it takes one argument"
  | some (AliasValue.binary _) => throw ↑s!"parser '{aliasName}' is not a constant, it takes two arguments"
  | none   => throw ↑s!"parser '{aliasName}' was not found"

def getUnaryAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (α → α) := do
  match (← getAlias mapRef aliasName) with
  | some (AliasValue.unary v) => pure v
  | some _ => throw ↑s!"parser '{aliasName}' does not take one argument"
  | none   => throw ↑s!"parser '{aliasName}' was not found"

def getBinaryAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (α → α → α) := do
  match (← getAlias mapRef aliasName) with
  | some (AliasValue.binary v) => pure v
  | some _ => throw ↑s!"parser '{aliasName}' does not take two arguments"
  | none   => throw ↑s!"parser '{aliasName}' was not found"

abbrev ParserAliasValue := AliasValue Parser

builtin_initialize parserAliasesRef : IO.Ref (NameMap ParserAliasValue) ← IO.mkRef {}

-- Later, we define macro registerParserAlias! which registers a parser, formatter and parenthesizer
def registerAlias (aliasName : Name) (p : ParserAliasValue) : IO Unit := do
```

```
  registerAliasCore parserAliasesRef aliasName p

instance : Coe Parser ParserAliasValue := { coe := AliasValue.const }
instance : Coe (Parser → Parser) ParserAliasValue := { coe := AliasValue.unary }
instance : Coe (Parser → Parser → Parser) ParserAliasValue := { coe := AliasValue.binary }

def isParserAlias (aliasName : Name) : IO Bool := do
  match (← getAlias parserAliasesRef aliasName) with
  | some _ => pure true
  | _      => pure false

def ensureUnaryParserAlias (aliasName : Name) : IO Unit :=
  discard $ getUnaryAlias parserAliasesRef aliasName

def ensureBinaryParserAlias (aliasName : Name) : IO Unit :=
  discard $ getBinaryAlias parserAliasesRef aliasName

def ensureConstantParserAlias (aliasName : Name) : IO Unit :=
  discard $ getConstAlias parserAliasesRef aliasName

partial def compileParserDescr (categories : ParserCategories) (d : ParserDescr) : ImportM Parser :=
  let rec visit : ParserDescr → ImportM Parser
    | ParserDescr.const n                             => getConstAlias parserAliasesRef n
    | ParserDescr.unary n d                           => return (← getUnaryAlias parserAliasesRef n) (← visit d)
    | ParserDescr.binary n d₁ d₂                      => return (← getBinaryAlias parserAliasesRef n) (← visit d₁) (← visit d₂)
    | ParserDescr.node k prec d                       => return leadingNode k prec (← visit d)
    | ParserDescr.nodeWithAntiquot n k d              => return nodeWithAntiquot n k (← visit d) (anonymous := true)
    | ParserDescr.sepBy p sep psep trail              => return sepBy (← visit p) sep (← visit psep) trail
    | ParserDescr.sepBy1 p sep psep trail             => return sepBy1 (← visit p) sep (← visit psep) trail
    | ParserDescr.trailingNode k prec lhsPrec d       => return trailingNode k prec lhsPrec (← visit d)
    | ParserDescr.symbol tk                           => return symbol tk
    | ParserDescr.nonReservedSymbol tk includeIdent   => return nonReservedSymbol tk includeIdent
    | ParserDescr.parser constName                    => do
      let (_, p) ← mkParserOfConstantAux categories constName visit;
      pure p
    | ParserDescr.cat catName prec                    =>
      match getCategory categories catName with
      | some _ => pure $ categoryParser catName prec
      | none   => IO.ofExcept $ throwUnknownParserCategory catName
  visit d

def mkParserOfConstant (categories : ParserCategories) (constName : Name) : ImportM (Bool × Parser) :=
  mkParserOfConstantAux categories constName (compileParserDescr categories)

structure ParserAttributeHook where
  /- Called after a parser attribute is applied to a declaration. -/
  postAdd (catName : Name) (declName : Name) (builtin : Bool) : AttrM Unit
```

```
builtin_initialize parserAttributeHooks : IO.Ref (List ParserAttributeHook) ← IO.mkRef {}

def registerParserAttributeHook (hook : ParserAttributeHook) : IO Unit := do
  parserAttributeHooks.modify fun hooks => hook::hooks

def runParserAttributeHooks (catName : Name) (declName : Name) (builtin : Bool) : AttrM Unit := do
  let hooks ← parserAttributeHooks.get
  hooks.forM fun hook => hook.postAdd catName declName builtin

builtin_initialize
  registerBuiltinAttribute {
    name  := `runBuiltinParserAttributeHooks,
    descr := "explicitly run hooks normally activated by builtin parser attributes",
    add   := fun decl stx persistent => do
      Attribute.Builtin.ensureNoArgs stx
      runParserAttributeHooks Name.anonymous decl (builtin := true)
  }

builtin_initialize
  registerBuiltinAttribute {
    name  := `runParserAttributeHooks,
    descr := "explicitly run hooks normally activated by parser attributes",
    add   := fun decl stx persistent => do
      Attribute.Builtin.ensureNoArgs stx
      runParserAttributeHooks Name.anonymous decl (builtin := false)
  }

private def ParserExtension.OLeanEntry.toEntry (s : State) : OLeanEntry → ImportM Entry
  | token tk     => return Entry.token tk
  | kind k       => return Entry.kind k
  | category c l => return Entry.category c l
  | parser catName declName prio => do
    let (leading, p) ← mkParserOfConstant s.categories declName
    Entry.parser catName declName leading p prio

builtin_initialize parserExtension : ParserExtension ←
  registerScopedEnvExtension {
    name            := `parserExt
    mkInitial       := ParserExtension.mkInitial
    addEntry        := ParserExtension.addEntryImpl
    toOLeanEntry    := ParserExtension.Entry.toOLeanEntry
    ofOLeanEntry    := ParserExtension.OLeanEntry.toEntry
  }

def isParserCategory (env : Environment) (catName : Name) : Bool :=
  (parserExtension.getState env).categories.contains catName
```

```
def addParserCategory (env : Environment) (catName : Name) (behavior : LeadingIdentBehavior) : Except String Environment := do
  if isParserCategory env catName then
    throwParserCategoryAlreadyDefined catName
  else
    return parserExtension.addEntry env <| ParserExtension.Entry.category catName behavior

def leadingIdentBehavior (env : Environment) (catName : Name) : LeadingIdentBehavior :=
  match getCategory (parserExtension.getState env).categories catName with
  | none     => LeadingIdentBehavior.default
  | some cat => cat.behavior

def mkCategoryAntiquotParser (kind : Name) : Parser :=
  mkAntiquot kind.toString none

-- helper decl to work around inlining issue https://github.com/leanprover/lean4/commit/3f6de2af06dd9a25f62294129f64bc05a29ea912#r41340
@[inline] private def mkCategoryAntiquotParserFn (kind : Name) : ParserFn :=
  (mkCategoryAntiquotParser kind).fn

def categoryParserFnImpl (catName : Name) : ParserFn := fun ctx s =>
  let catName := if catName == `syntax then `stx else catName -- temporary Hack
  let categories := (parserExtension.getState ctx.env).categories
  match getCategory categories catName with
  | some cat =>
    prattParser catName cat.tables cat.behavior (mkCategoryAntiquotParserFn catName) ctx s
  | none     => s.mkUnexpectedError ("unknown parser category '" ++ toString catName ++ "'")

@[builtinInit] def setCategoryParserFnRef : IO Unit :=
  categoryParserFnRef.set categoryParserFnImpl

def addToken (tk : Token) (kind : AttributeKind) : AttrM Unit := do
  -- Recall that `ParserExtension.addEntry` is pure, and assumes `addTokenConfig` does not fail.
  -- So, we must run it here to handle exception.
  discard <| ofExcept <| addTokenConfig (parserExtension.getState (← getEnv)).tokens tk
  parserExtension.add (ParserExtension.Entry.token tk) kind

def addSyntaxNodeKind (env : Environment) (k : SyntaxNodeKind) : Environment :=
  parserExtension.addEntry env <| ParserExtension.Entry.kind k

def isValidSyntaxNodeKind (env : Environment) (k : SyntaxNodeKind) : Bool :=
  let kinds := (parserExtension.getState env).kinds
  kinds.contains k

def getSyntaxNodeKinds (env : Environment) : List SyntaxNodeKind := do
  let kinds := (parserExtension.getState env).kinds
  kinds.foldl (fun ks k _ => k::ks) []
```

```
def getTokenTable (env : Environment) : TokenTable :=
  (parserExtension.getState env).tokens

def mkInputContext (input : String) (fileName : String) : InputContext := {
  input    := input,
  fileName := fileName,
  fileMap  := input.toFileMap
}

def mkParserContext (ictx : InputContext) (pmctx : ParserModuleContext) : ParserContext := {
  prec                := 0,
  toInputContext      := ictx,
  toParserModuleContext := pmctx,
  tokens              := getTokenTable pmctx.env
}

def mkParserState (input : String) : ParserState :=
  { cache := initCacheForInput input }

/- convenience function for testing -/
def runParserCategory (env : Environment) (catName : Name) (input : String) (fileName := "<input>") : Except String Syntax :=
  let c := mkParserContext (mkInputContext input fileName) { env := env, options := {} }
  let s := mkParserState input
  let s := whitespace c s
  let s := categoryParserFnImpl catName c s
  if s.hasError then
    Except.error (s.toErrorMsg c)
  else if input.atEnd s.pos then
    Except.ok s.stxStack.back
  else
    Except.error ((s.mkError "end of input").toErrorMsg c)

def declareBuiltinParser (env : Environment) (addFnName : Name) (catName : Name) (declName : Name) (prio : Nat) : IO Environment :=
  let name := `_regBuiltinParser ++ declName
  let type := mkApp (mkConst `IO) (mkConst `Unit)
  let val  := mkAppN (mkConst addFnName) #[toExpr catName, toExpr declName, mkConst declName, mkNatLit prio]
  let decl := Declaration.defnDecl { name := name, levelParams := [], type := type, value := val, hints := ReducibilityHints.opaque,
                                     safety := DefinitionSafety.safe }
  match env.addAndCompile {} decl with
  -- TODO: pretty print error
  | Except.error _ => throw (IO.userError ("failed to emit registration code for builtin parser '" ++ toString declName ++ "'"))
  | Except.ok env  => IO.ofExcept (setBuiltinInitAttr env name)

def declareLeadingBuiltinParser (env : Environment) (catName : Name) (declName : Name) (prio : Nat) : IO Environment := -- TODO: use Co
  declareBuiltinParser env `Lean.Parser.addBuiltinLeadingParser catName declName prio

def declareTrailingBuiltinParser (env : Environment) (catName : Name) (declName : Name) (prio : Nat) : IO Environment := -- TODO: use C
```

```lean
  declareBuiltinParser env `Lean.Parser.addBuiltinTrailingParser catName declName prio

def getParserPriority (args : Syntax) : Except String Nat :=
  match args.getNumArgs with
  | 0 => pure 0
  | 1 => match (args.getArg 0).isNatLit? with
    | some prio => pure prio
    | none => throw "invalid parser attribute, numeral expected"
  | _ => throw "invalid parser attribute, no argument or numeral expected"

private def BuiltinParserAttribute.add (attrName : Name) (catName : Name)
    (declName : Name) (stx : Syntax) (kind : AttributeKind) : AttrM Unit := do
  let prio ← Attribute.Builtin.getPrio stx
  unless kind == AttributeKind.global do throwError "invalid attribute '{attrName}', must be global"
  let decl ← getConstInfo declName
  let env ← getEnv
  match decl.type with
  | Expr.const `Lean.Parser.TrailingParser _ _ => do
    let env ← declareTrailingBuiltinParser env catName declName prio
    setEnv env
  | Expr.const `Lean.Parser.Parser _ _ => do
    let env ← declareLeadingBuiltinParser env catName declName prio
    setEnv env
  | _ => throwError "unexpected parser type at '{declName}' (`Parser` or `TrailingParser` expected)"
  runParserAttributeHooks catName declName (builtin := true)

/-
The parsing tables for builtin parsers are "stored" in the extracted source code.
-/
def registerBuiltinParserAttribute (attrName : Name) (catName : Name) (behavior := LeadingIdentBehavior.default) : IO Unit := do
  addBuiltinParserCategory catName behavior
  registerBuiltinAttribute {
   name            := attrName,
   descr           := "Builtin parser",
   add             := fun declName stx kind => liftM $ BuiltinParserAttribute.add attrName catName declName stx kind,
   applicationTime := AttributeApplicationTime.afterCompilation
  }

private def ParserAttribute.add (attrName : Name) (catName : Name) (declName : Name) (stx : Syntax) (attrKind : AttributeKind) : AttrM
  let prio ← Attribute.Builtin.getPrio stx
  let env ← getEnv
  let opts ← getOptions
  let categories := (parserExtension.getState env).categories
  let p ← mkParserOfConstant categories declName
  let leading    := p.1
  let parser     := p.2
  let tokens     := parser.info.collectTokens []
```

```
  tokens.forM fun token => do
    try
      addToken token attrKind
    catch
      | Exception.error ref msg => throwError "invalid parser '{declName}', {msg}"
      | ex => throw ex
  let kinds := parser.info.collectKinds {}
  kinds.forM fun kind _ => modifyEnv fun env => addSyntaxNodeKind env kind
  let entry := ParserExtension.Entry.parser catName declName leading parser prio
  match addParser categories catName declName leading parser prio with
  | Except.error ex => throwError ex
  | Except.ok _     => parserExtension.add entry attrKind
  runParserAttributeHooks catName declName (builtin := false)

def mkParserAttributeImpl (attrName : Name) (catName : Name) : AttributeImpl where
  name                      := attrName
  descr                     := "parser"
  add declName stx attrKind := ParserAttribute.add attrName catName declName stx attrKind
  applicationTime           := AttributeApplicationTime.afterCompilation

/- A builtin parser attribute that can be extended by users. -/
def registerBuiltinDynamicParserAttribute (attrName : Name) (catName : Name) : IO Unit := do
  registerBuiltinAttribute (mkParserAttributeImpl attrName catName)

@[builtinInit] private def registerParserAttributeImplBuilder : IO Unit :=
  registerAttributeImplBuilder `parserAttr fun args =>
    match args with
    | [DataValue.ofName attrName, DataValue.ofName catName] => pure $ mkParserAttributeImpl attrName catName
    | _ => throw "invalid parser attribute implementation builder arguments"

def registerParserCategory (env : Environment) (attrName : Name) (catName : Name) (behavior := LeadingIdentBehavior.default) : IO Envir
  let env ← IO.ofExcept $ addParserCategory env catName behavior
  registerAttributeOfBuilder env `parserAttr [DataValue.ofName attrName, DataValue.ofName catName]

-- declare `termParser` here since it is used everywhere via antiquotations

builtin_initialize registerBuiltinParserAttribute `builtinTermParser `term

builtin_initialize registerBuiltinDynamicParserAttribute `termParser `term

-- declare `commandParser` to break cyclic dependency
builtin_initialize registerBuiltinParserAttribute `builtinCommandParser `command

builtin_initialize registerBuiltinDynamicParserAttribute `commandParser `command

@[inline] def commandParser (rbp : Nat := 0) : Parser :=
  categoryParser `command rbp
```

```
def notFollowedByCategoryTokenFn (catName : Name) : ParserFn := fun ctx s =>
  let categories := (parserExtension.getState ctx.env).categories
  match getCategory categories catName with
  | none => s.mkUnexpectedError s!"unknown parser category '{catName}'"
  | some cat =>
    let (s, stx) := peekToken ctx s
    match stx with
    | Except.ok (Syntax.atom _ sym) =>
      if ctx.insideQuot && sym == "$" then s
      else match cat.tables.leadingTable.find? (Name.mkSimple sym) with
      | some _ => s.mkUnexpectedError (toString catName)
      | _      => s
    | Except.ok _    => s
    | Except.error _ => s

@[inline] def notFollowedByCategoryToken (catName : Name) : Parser := {
  fn := notFollowedByCategoryTokenFn catName
}

abbrev notFollowedByCommandToken : Parser :=
  notFollowedByCategoryToken `command

abbrev notFollowedByTermToken : Parser :=
  notFollowedByCategoryToken `term

end Parser
end Lean
::::::::::::::
Parser/Extra.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Extension
-- necessary for auto-generation
import Lean.PrettyPrinter.Parenthesizer
import Lean.PrettyPrinter.Formatter

namespace Lean
namespace Parser

-- synthesize pretty printers for parsers declared prior to `Lean.PrettyPrinter`
-- (because `Parser.Extension` depends on them)
attribute [runBuiltinParserAttributeHooks]
```

```
  leadingNode termParser commandParser mkAntiquot nodeWithAntiquot sepBy sepBy1
  unicodeSymbol nonReservedSymbol

@[runBuiltinParserAttributeHooks] def optional (p : Parser) : Parser :=
  optionalNoAntiquot (withAntiquotSpliceAndSuffix `optional p (symbol "?"))

@[runBuiltinParserAttributeHooks] def many (p : Parser) : Parser :=
  manyNoAntiquot (withAntiquotSpliceAndSuffix `many p (symbol "*"))

@[runBuiltinParserAttributeHooks] def many1 (p : Parser) : Parser :=
  many1NoAntiquot (withAntiquotSpliceAndSuffix `many p (symbol "*"))

@[runBuiltinParserAttributeHooks] def ident : Parser :=
  withAntiquot (mkAntiquot "ident" identKind) identNoAntiquot

-- `ident` and `rawIdent` produce the same syntax tree, so we reuse the antiquotation kind name
@[runBuiltinParserAttributeHooks] def rawIdent : Parser :=
  withAntiquot (mkAntiquot "ident" identKind) rawIdentNoAntiquot

@[runBuiltinParserAttributeHooks] def numLit : Parser :=
  withAntiquot (mkAntiquot "numLit" numLitKind) numLitNoAntiquot

@[runBuiltinParserAttributeHooks] def scientificLit : Parser :=
  withAntiquot (mkAntiquot "scientificLit" scientificLitKind) scientificLitNoAntiquot

@[runBuiltinParserAttributeHooks] def strLit : Parser :=
  withAntiquot (mkAntiquot "strLit" strLitKind) strLitNoAntiquot

@[runBuiltinParserAttributeHooks] def charLit : Parser :=
  withAntiquot (mkAntiquot "charLit" charLitKind) charLitNoAntiquot

@[runBuiltinParserAttributeHooks] def nameLit : Parser :=
  withAntiquot (mkAntiquot "nameLit" nameLitKind) nameLitNoAntiquot

@[runBuiltinParserAttributeHooks, inline] def group (p : Parser) : Parser :=
  node groupKind p

@[runBuiltinParserAttributeHooks, inline] def many1Indent (p : Parser) : Parser :=
  withPosition $ many1 (checkColGe "irrelevant" >> p)

@[runBuiltinParserAttributeHooks, inline] def manyIndent (p : Parser) : Parser :=
  withPosition $ many (checkColGe "irrelevant" >> p)

@[runBuiltinParserAttributeHooks] abbrev notSymbol (s : String) : Parser :=
  notFollowedBy (symbol s) s

/-- No-op parser that advises the pretty printer to emit a non-breaking space. -/
```

```
@[inline] def ppHardSpace : Parser := skip
/-- No-op parser that advises the pretty printer to emit a space/soft line break. -/
@[inline] def ppSpace : Parser := skip
/-- No-op parser that advises the pretty printer to emit a hard line break. -/
@[inline] def ppLine : Parser := skip
/--
  No-op parser combinator that advises the pretty printer to group and indent the given syntax.
  By default, only syntax categories are grouped. -/
@[inline] def ppGroup : Parser → Parser := id
/-- No-op parser combinator that advises the pretty printer to indent the given syntax without grouping it. -/
@[inline] def ppIndent : Parser → Parser := id
/--
  No-op parser combinator that advises the pretty printer to dedent the given syntax.
  Dedenting can in particular be used to counteract automatic indentation. -/
@[inline] def ppDedent : Parser → Parser := id

end Parser

section
open PrettyPrinter

@[combinatorFormatter Lean.Parser.ppHardSpace] def ppHardSpace.formatter : Formatter := Formatter.push " "
@[combinatorFormatter Lean.Parser.ppSpace] def ppSpace.formatter : Formatter := Formatter.pushLine
@[combinatorFormatter Lean.Parser.ppLine] def ppLine.formatter : Formatter := Formatter.push "\n"
@[combinatorFormatter Lean.Parser.ppGroup] def ppGroup.formatter (p : Formatter) : Formatter := Formatter.group $ Formatter.indent p
@[combinatorFormatter Lean.Parser.ppIndent] def ppIndent.formatter (p : Formatter) : Formatter := Formatter.indent p
@[combinatorFormatter Lean.Parser.ppDedent] def ppDedent.formatter (p : Formatter) : Formatter := do
  let opts ← getOptions
  Formatter.indent p (some ((0:Int) - Std.Format.getIndent opts))
end

namespace Parser

-- now synthesize parenthesizers
attribute [runBuiltinParserAttributeHooks]
  ppHardSpace ppSpace ppLine ppGroup ppIndent ppDedent

macro "register_parser_alias" aliasName:strLit declName:ident : term =>
  `(do Parser.registerAlias $aliasName $declName
       PrettyPrinter.Formatter.registerAlias $aliasName $(mkIdentFrom declName (declName.getId ++ `formatter))
       PrettyPrinter.Parenthesizer.registerAlias $aliasName $(mkIdentFrom declName (declName.getId ++ `parenthesizer)))

builtin_initialize
  register_parser_alias "group" group
  register_parser_alias "ppHardSpace" ppHardSpace
  register_parser_alias "ppSpace" ppSpace
  register_parser_alias "ppLine" ppLine
```

```
  register_parser_alias "ppGroup" ppGroup
  register_parser_alias "ppIndent" ppIndent
  register_parser_alias "ppDedent" ppDedent

end Parser

open Parser

open PrettyPrinter.Parenthesizer (registerAlias) in
builtin_initialize
  registerAlias "num" numLit.parenthesizer
  registerAlias "scientific" scientificLit.parenthesizer
  registerAlias "str" strLit.parenthesizer
  registerAlias "char" charLit.parenthesizer
  registerAlias "name" nameLit.parenthesizer
  registerAlias "ident" ident.parenthesizer
  registerAlias "many" many.parenthesizer
  registerAlias "many1" many1.parenthesizer
  registerAlias "optional" optional.parenthesizer

open PrettyPrinter.Formatter (registerAlias) in
builtin_initialize
  registerAlias "num" numLit.formatter
  registerAlias "scientific" scientificLit.formatter
  registerAlias "str" strLit.formatter
  registerAlias "char" charLit.formatter
  registerAlias "name" nameLit.formatter
  registerAlias "ident" ident.formatter
  registerAlias "many" many.formatter
  registerAlias "many1" many1.formatter
  registerAlias "optional" optional.formatter

end Lean
::::::::::::::
Parser/Level.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Extra

namespace Lean
namespace Parser

builtin_initialize
```

```
  registerBuiltinParserAttribute `builtinLevelParser `level

@[inline] def levelParser (rbp : Nat := 0) : Parser :=
  categoryParser `level rbp

namespace Level

@[builtinLevelParser] def paren  := leading_parser "(" >> levelParser >> ")"
@[builtinLevelParser] def max    := leading_parser nonReservedSymbol "max" true  >> many1 (ppSpace >> levelParser maxPrec)
@[builtinLevelParser] def imax   := leading_parser nonReservedSymbol "imax" true >> many1 (ppSpace >> levelParser maxPrec)
@[builtinLevelParser] def hole   := leading_parser "_"
@[builtinLevelParser] def num    := checkPrec maxPrec >> numLit
@[builtinLevelParser] def ident  := checkPrec maxPrec >> Parser.ident
@[builtinLevelParser] def addLit := trailing_parser:65 " + " >> numLit

end Level

end Parser
end Lean
::::::::::::::
Parser/Module.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Message
import Lean.Parser.Command

namespace Lean
namespace Parser

namespace Module
def «prelude»  := leading_parser "prelude"
def «import»   := leading_parser "import " >> optional "runtime" >> ident
def header     := leading_parser optional («prelude» >> ppLine) >> many («import» >> ppLine) >> ppLine
/--
  Parser for a Lean module. We never actually run this parser but instead use the imperative definitions below that
  return the same syntax tree structure, but add error recovery. Still, it is helpful to have a `Parser` definition
  for it in order to auto-generate helpers such as the pretty printer. -/
@[runBuiltinParserAttributeHooks]
def module     := leading_parser header >> many (commandParser >> ppLine >> ppLine)

def updateTokens (c : ParserContext) : ParserContext :=
  { c with
    tokens := match addParserTokens c.tokens header.info with
```

```
    | Except.ok tables => tables
    | Except.error _   => unreachable! }

end Module

structure ModuleParserState where
  pos        : String.Pos := 0
  recovering : Bool       := false
  deriving Inhabited

private def mkErrorMessage (c : ParserContext) (pos : String.Pos) (errorMsg : String) : Message :=
  let pos := c.fileMap.toPosition pos
  { fileName := c.fileName, pos := pos, data := errorMsg }

def parseHeader (inputCtx : InputContext) : IO (Syntax × ModuleParserState × MessageLog) := do
  let dummyEnv ← mkEmptyEnvironment
  let ctx := mkParserContext inputCtx { env := dummyEnv, options := {} }
  let ctx := Module.updateTokens ctx
  let s   := mkParserState ctx.input
  let s   := whitespace ctx s
  let s   := Module.header.fn ctx s
  let stx := s.stxStack.back
  match s.errorMsg with
  | some errorMsg =>
    let msg := mkErrorMessage ctx s.pos (toString errorMsg)
    pure (stx, { pos := s.pos, recovering := true }, { : MessageLog }.add msg)
  | none =>
    pure (stx, { pos := s.pos }, {})

private def mkEOI (pos : String.Pos) : Syntax :=
  let atom := mkAtom (SourceInfo.original "".toSubstring pos "".toSubstring) ""
  Syntax.node `Lean.Parser.Module.eoi #[atom]

def isEOI (s : Syntax) : Bool :=
  s.isOfKind `Lean.Parser.Module.eoi

def isExitCommand (s : Syntax) : Bool :=
  s.isOfKind `Lean.Parser.Command.exit

private def consumeInput (c : ParserContext) (pos : String.Pos) : String.Pos :=
  let s : ParserState := { cache := initCacheForInput c.input, pos := pos }
  let s := tokenFn [] c s
  match s.errorMsg with
  | some _ => pos + 1
  | none   => s.pos

def topLevelCommandParserFn : ParserFn :=
```

```
  commandParser.fn

partial def parseCommand (inputCtx : InputContext) (pmctx : ParserModuleContext) (s : ModuleParserState) (messages : MessageLog) : Synt
  let rec parse (s : ModuleParserState) (messages : MessageLog) :=
    let { pos := pos, recovering := recovering } := s
    if inputCtx.input.atEnd pos then
      (mkEOI pos, s, messages)
    else
      let c   := mkParserContext inputCtx pmctx
      let s   := { cache := initCacheForInput c.input, pos := pos : ParserState }
      let s   := whitespace c s
      let s   := topLevelCommandParserFn c s
      match s.errorMsg with
      | none =>
        (s.stxStack.back, { pos := s.pos }, messages)
      | some errorMsg =>
        -- advance at least one token to prevent infinite loops
        let pos := if s.pos == pos then consumeInput c s.pos else s.pos
        /- We ignore commands where `getPos?` is none. This happens only on commands that have a prefix comprised of optional elements.
           For example, unification hints start with `optional («scoped» <|> «local»)`.
           We claim a syntactically incorrect command containing no token or identifier is irrelevant for intellisense and should be ig
        let ignore := s.stxStack.isEmpty || s.stxStack.back.getPos?.isNone
        let messages := if recovering && ignore then messages else messages.add <| mkErrorMessage c s.pos (toString errorMsg)
        if ignore then
          parse { pos := pos, recovering := true } messages
        else
          (s.stxStack.back, { pos := pos, recovering := true }, messages)
  parse s messages

-- only useful for testing since most Lean files cannot be parsed without elaboration

partial def testParseModuleAux (env : Environment) (inputCtx : InputContext) (s : ModuleParserState) (msgs : MessageLog) (stxs  : Array
  let rec parse (state : ModuleParserState) (msgs : MessageLog) (stxs : Array Syntax) :=
    match parseCommand inputCtx { env := env, options := {} } state msgs with
    | (stx, state, msgs) =>
      if isEOI stx then
        if msgs.isEmpty then
          pure stxs
        else do
          msgs.forM fun msg => msg.toString >>= IO.println
          throw (IO.userError "failed to parse file")
      else
        parse state msgs (stxs.push stx)
  parse s msgs stxs

def testParseModule (env : Environment) (fname contents : String) : IO Syntax := do
  let fname ← IO.realPath fname
```

```
  let inputCtx := mkInputContext contents fname
  let (header, state, messages) ← parseHeader inputCtx
  let cmds ← testParseModuleAux env inputCtx state messages #[]
  let stx := Syntax.node `Lean.Parser.Module.module #[header, mkListNode cmds]
  pure stx.updateLeading

def testParseFile (env : Environment) (fname : String) : IO Syntax := do
  let contents ← IO.FS.readFile fname
  testParseModule env fname contents

end Parser
end Lean
::::::::::::::
Parser/StrInterpolation.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Parser.Basic
namespace Lean.Parser

def isQuotableCharForStrInterpolant (c : Char) : Bool :=
  c == '{' || isQuotableCharDefault c

partial def interpolatedStrFn (p : ParserFn) : ParserFn := fun c s =>
  let input     := c.input
  let stackSize := s.stackSize
  let rec parse (startPos : Nat) (c : ParserContext) (s : ParserState) : ParserState :=
    let i     := s.pos
    if input.atEnd i then
      s.mkUnexpectedErrorAt "unterminated string literal" startPos
    else
      let curr := input.get i
      let s    := s.setPos (input.next i)
      if curr == '\"' then
        let s := mkNodeToken interpolatedStrLitKind startPos c s
        s.mkNode interpolatedStrKind stackSize
      else if curr == '\\' then
        andthenFn (quotedCharCoreFn isQuotableCharForStrInterpolant) (parse startPos) c s
      else if curr == '{' then
        let s := mkNodeToken interpolatedStrLitKind startPos c s
        let s := p c s
        if s.hasError then s
        else
          let pos := s.pos
```

```
          andthenFn (satisfyFn (· == '}') "expected '}'") (parse pos) c s
      else
        parse startPos c s
  let startPos := s.pos
  if input.atEnd startPos then
    s.mkEOIError
  else
    let curr  := input.get s.pos;
    if curr != '\"' then
      s.mkError "interpolated string"
    else
      let s := s.next input startPos
      parse startPos c s

@[inline] def interpolatedStrNoAntiquot (p : Parser) : Parser := {
  fn   := interpolatedStrFn p.fn,
  info := mkAtomicInfo "interpolatedStr"
}

def interpolatedStr (p : Parser) : Parser :=
  withAntiquot (mkAntiquot "interpolatedStr" interpolatedStrKind) $ interpolatedStrNoAntiquot p

end Lean.Parser
::::::::::::::
Parser/Syntax.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Command
import Lean.Parser.Tactic

namespace Lean
namespace Parser

builtin_initialize
  registerBuiltinParserAttribute `builtinSyntaxParser `stx LeadingIdentBehavior.both
  registerBuiltinDynamicParserAttribute `stxParser `stx

builtin_initialize
  registerBuiltinParserAttribute `builtinPrecParser `prec LeadingIdentBehavior.both
  registerBuiltinDynamicParserAttribute `precParser `prec

@[inline] def precedenceParser (rbp : Nat := 0) : Parser :=
  categoryParser `prec rbp
```

```
@[inline] def syntaxParser (rbp : Nat := 0) : Parser :=
  categoryParser `stx rbp

def «precedence» := leading_parser ":" >> precedenceParser maxPrec
def optPrecedence := optional (atomic «precedence»)

namespace Syntax
@[builtinPrecParser] def numPrec := checkPrec maxPrec >> numLit

@[builtinSyntaxParser] def paren           := leading_parser "(" >> many1 syntaxParser >> ")"
@[builtinSyntaxParser] def cat             := leading_parser ident >> optPrecedence
@[builtinSyntaxParser] def unary           := leading_parser ident >> checkNoWsBefore >> "(" >> many1 syntaxParser >> ")"
@[builtinSyntaxParser] def binary          := leading_parser ident >> checkNoWsBefore >> "(" >> many1 syntaxParser >> ", " >> many1 syn
@[builtinSyntaxParser] def sepBy           := leading_parser "sepBy(" >> many1 syntaxParser >> ", " >> strLit >> optional (", " >> many
@[builtinSyntaxParser] def sepBy1          := leading_parser "sepBy1(" >> many1 syntaxParser >> ", " >> strLit >> optional (", " >> man
@[builtinSyntaxParser] def atom            := leading_parser strLit
@[builtinSyntaxParser] def nonReserved     := leading_parser "&" >> strLit

end Syntax

namespace Term

@[builtinTermParser] def stx.quot : Parser := leading_parser "`(stx|"  >> toggleInsideQuot syntaxParser >> ")"
@[builtinTermParser] def prec.quot : Parser := leading_parser "`(prec|"  >> toggleInsideQuot precedenceParser >> ")"
@[builtinTermParser] def prio.quot : Parser := leading_parser "`(prio|"  >> toggleInsideQuot priorityParser >> ")"

end Term

namespace Command

def namedName := leading_parser (atomic ("(" >> nonReservedSymbol "name") >> " := " >> ident >> ")")
def optNamedName := optional namedName

def «prefix»   := leading_parser "prefix"
def «infix»    := leading_parser "infix"
def «infixl»   := leading_parser "infixl"
def «infixr»   := leading_parser "infixr"
def «postfix»  := leading_parser "postfix"
def mixfixKind := «prefix» <|> «infix» <|> «infixl» <|> «infixr» <|> «postfix»
@[builtinCommandParser] def «mixfix»   := leading_parser Term.attrKind >> mixfixKind >> optPrecedence >> optNamedName >> optNamedPrio >>
-- NOTE: We use `suppressInsideQuot` in the following parsers because quotations inside them are evaluated in the same stage and
-- thus should be ignored when we use `checkInsideQuot` to prepare the next stage for a builtin syntax change
def identPrec  := leading_parser ident >> optPrecedence

def optKind : Parser := optional ("(" >> nonReservedSymbol "kind" >> ":=" >> ident >> ")")

```

```
def notationItem := ppSpace >> withAntiquot (mkAntiquot "notationItem" `Lean.Parser.Command.notationItem) (strLit <|> identPrec)
@[builtinCommandParser] def «notation»    := leading_parser Term.attrKind >> "notation" >> optPrecedence >> optNamedName >> optNamedPri
@[builtinCommandParser] def «macro_rules» := suppressInsideQuot (leading_parser "macro_rules" >>  optKind >> Term.matchAlts)
@[builtinCommandParser] def «syntax»      := leading_parser Term.attrKind >> "syntax " >> optPrecedence >> optNamedName >> optNamedPrio
@[builtinCommandParser] def syntaxAbbrev  := leading_parser "syntax " >> ident >> " := " >> many1 syntaxParser
@[builtinCommandParser] def syntaxCat     := leading_parser "declare_syntax_cat " >> ident
def macroArgSimple := leading_parser ident >> checkNoWsBefore "no space before ':'" >> ":" >> syntaxParser maxPrec
def macroArgSymbol := leading_parser strLit >> optional (atomic <| checkNoWsBefore >> "%" >> checkNoWsBefore >> ident)
def macroArg  := macroArgSymbol <|> atomic macroArgSimple
def macroHead := macroArg
def macroTailTactic   : Parser := atomic (" : " >> identEq "tactic") >> darrow >> ("`(" >> toggleInsideQuot Tactic.seq1 >> ")" <|> term
def macroTailCommand  : Parser := atomic (" : " >> identEq "command") >> darrow >> ("`(" >> toggleInsideQuot (many1Unbox commandParser)
def macroTailDefault  : Parser := atomic (" : " >> ident) >> darrow >> (("`(" >> toggleInsideQuot (categoryParserOfStack 2) >> ")") <|>
def macroTail := macroTailTactic <|> macroTailCommand <|> macroTailDefault
@[builtinCommandParser] def «macro»       := leading_parser suppressInsideQuot (Term.attrKind >> "macro " >> optPrecedence >> optNamedN

@[builtinCommandParser] def «elab_rules» := leading_parser suppressInsideQuot ("elab_rules" >> optKind >> optional (" : " >> ident) >> 
def elabHead := macroHead
def elabArg  := macroArg
def elabTail := atomic (" : " >> ident >> optional (" <= " >> ident)) >> darrow >> termParser
@[builtinCommandParser] def «elab»       := leading_parser suppressInsideQuot (Term.attrKind >> "elab " >> optPrecedence >> optNamedNam

end Command

end Parser
end Lean
::::::::::::::
Parser/Tactic.lean
::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Term

namespace Lean
namespace Parser
namespace Tactic

builtin_initialize
  register_parser_alias "tacticSeq"    tacticSeq

@[builtinTacticParser] def «unknown»    := leading_parser withPosition (ident >> errorAtSavedPos "unknown tactic" true)
@[builtinTacticParser] def nestedTactic := tacticSeqBracketed

/- Auxiliary parser for expanding `match` tactic -/
```

```
@[builtinTacticParser] def eraseAuxDiscrs := leading_parser:maxPrec "eraseAuxDiscrs!"

def matchRhs  := Term.hole <|> Term.syntheticHole <|> tacticSeq
def matchAlts := Term.matchAlts (rhsParser := matchRhs)
@[builtinTacticParser] def «match» := leading_parser:leadPrec "match " >> optional Term.generalizingParam >> sepBy1 Term.matchDiscr ",
@[builtinTacticParser] def introMatch := leading_parser nonReservedSymbol "intro " >> matchAlts

@[builtinTacticParser] def decide := leading_parser nonReservedSymbol "decide"
@[builtinTacticParser] def nativeDecide := leading_parser nonReservedSymbol "nativeDecide"

end Tactic
end Parser
end Lean
::::::::::::::
Parser/Term.lean
::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura, Sebastian Ullrich
-/
import Lean.Parser.Attr
import Lean.Parser.Level

namespace Lean
namespace Parser

namespace Command
def commentBody : Parser :=
{ fn := rawFn (fun c s => finishCommentBlock s.pos 1 c s) (trailingWs := true) }

@[combinatorParenthesizer Lean.Parser.Command.commentBody] def commentBody.parenthesizer := PrettyPrinter.Parenthesizer.visitToken
@[combinatorFormatter Lean.Parser.Command.commentBody] def commentBody.formatter := PrettyPrinter.Formatter.visitAtom Name.anonymous

def docComment       := leading_parser ppDedent $ "/--" >> commentBody >> ppLine
end Command

builtin_initialize
  registerBuiltinParserAttribute `builtinTacticParser `tactic LeadingIdentBehavior.both
  registerBuiltinDynamicParserAttribute `tacticParser `tactic

@[inline] def tacticParser (rbp : Nat := 0) : Parser :=
  categoryParser `tactic rbp

namespace Tactic

def tacticSeq1Indented : Parser :=
```

```
  leading_parser many1Indent (group (ppLine >> tacticParser >> optional ";"))
def tacticSeqBracketed : Parser :=
  leading_parser "{" >> many (group (ppLine >> tacticParser >> optional ";")) >> ppDedent (ppLine >> "}")
def tacticSeq :=
  nodeWithAntiquot "tacticSeq" `Lean.Parser.Tactic.tacticSeq (tacticSeqBracketed <|> tacticSeq1Indented)

/- Raw sequence for quotation and grouping -/
def seq1 :=
  node `Lean.Parser.Tactic.seq1 $ sepBy1 tacticParser ";\n" (allowTrailingSep := true)

end Tactic

def darrow : Parser := " => "

namespace Term

/- Built-in parsers -/

@[builtinTermParser] def byTactic := leading_parser:leadPrec "by " >> Tactic.tacticSeq

def optSemicolon (p : Parser) : Parser := ppDedent $ optional ";" >> ppLine >> p

-- `checkPrec` necessary for the pretty printer
@[builtinTermParser] def ident := checkPrec maxPrec >> Parser.ident
@[builtinTermParser] def num : Parser := checkPrec maxPrec >> numLit
@[builtinTermParser] def scientific : Parser := checkPrec maxPrec >> scientificLit
@[builtinTermParser] def str : Parser := checkPrec maxPrec >> strLit
@[builtinTermParser] def char : Parser := checkPrec maxPrec >> charLit
@[builtinTermParser] def type := leading_parser "Type" >> optional (checkWsBefore "" >> checkPrec leadPrec >> checkColGt >> levelParser
@[builtinTermParser] def sort := leading_parser "Sort" >> optional (checkWsBefore "" >> checkPrec leadPrec >> checkColGt >> levelParser
@[builtinTermParser] def prop := leading_parser "Prop"
@[builtinTermParser] def hole := leading_parser "_"
@[builtinTermParser] def syntheticHole := leading_parser "?" >> (ident <|> hole)
@[builtinTermParser] def «sorry» := leading_parser "sorry"
@[builtinTermParser] def cdot   := leading_parser symbol "·" <|> "."
@[builtinTermParser] def emptyC := leading_parser "∅" <|> (symbol "{" >> "}")
def typeAscription := leading_parser " : " >> termParser
def tupleTail      := leading_parser ", " >> sepBy1 termParser ", "
def parenSpecial : Parser := optional (tupleTail <|> typeAscription)
@[builtinTermParser] def paren := leading_parser "(" >> ppDedent (withoutPosition (withoutForbidden (optional (termParser >> parenSpeci
@[builtinTermParser] def anonymousCtor := leading_parser "⟨" >> sepBy termParser ", " >> "⟩"
def optIdent : Parser := optional (atomic (ident >> " : "))
def fromTerm   := leading_parser " from " >> termParser
def haveAssign := leading_parser " := " >> termParser
def haveDecl   := leading_parser optIdent >> termParser >> (haveAssign <|> fromTerm <|> byTactic)
@[builtinTermParser] def «have» := leading_parser:leadPrec withPosition ("have " >> haveDecl) >> optSemicolon termParser
def sufficesDecl := leading_parser optIdent >> termParser >> (fromTerm <|> byTactic)
```

```
@[builtinTermParser] def «suffices» := leading_parser:leadPrec withPosition ("suffices " >> sufficesDecl) >> optSemicolon termParser
@[builtinTermParser] def «show»     := leading_parser:leadPrec "show " >> termParser >> (fromTerm <|> byTactic)
def structInstArrayRef := leading_parser "[" >> termParser >>"]"
def structInstLVal   := leading_parser (ident <|> fieldIdx <|> structInstArrayRef) >> many (group ("." >> (ident <|> fieldIdx)) <|> str
def structInstField  := ppGroup $ leading_parser structInstLVal >> " := " >> termParser
def optEllipsis      := leading_parser optional ".."
@[builtinTermParser] def structInst := leading_parser "{" >> ppHardSpace >> optional (atomic (termParser >> " with "))
  >> manyIndent (group (structInstField >> optional ", "))
  >> optEllipsis
  >> optional (" : " >> termParser) >> " }"
def typeSpec := leading_parser " : " >> termParser
def optType : Parser := optional typeSpec
@[builtinTermParser] def explicit := leading_parser "@" >> termParser maxPrec
@[builtinTermParser] def inaccessible := leading_parser ".(" >> termParser >> ")"
def binderIdent : Parser  := ident <|> hole
def binderType (requireType := false) : Parser := if requireType then node nullKind (" : " >> termParser) else optional (" : " >> termP
def binderTactic  := leading_parser atomic (symbol " := " >> " by ") >> Tactic.tacticSeq
def binderDefault := leading_parser " := " >> termParser
def explicitBinder (requireType := false) := ppGroup $ leading_parser "(" >> many1 binderIdent >> binderType requireType >> optional (b
def implicitBinder (requireType := false) := ppGroup $ leading_parser "{" >> many1 binderIdent >> binderType requireType >> "}"
def instBinder := ppGroup $ leading_parser "[" >> optIdent >> termParser >> "]"
def bracketedBinder (requireType := false) := withAntiquot (mkAntiquot "bracketedBinder" none (anonymous := false)) <|
  explicitBinder requireType <|> implicitBinder requireType <|> instBinder

/-
It is feasible to support dependent arrows such as `{α} → α → α` without sacrificing the quality of the error messages for the longer c
`{α} → α → α` would be short for `{α : Type} → α → α`
Here is the encoding:
```
def implicitShortBinder := node `Lean.Parser.Term.implicitBinder $ "{" >> many1 binderIdent >> pushNone >> "}"
def depArrowShortPrefix := try (implicitShortBinder >> unicodeSymbol " → " " -> ")
def depArrowLongPrefix := bracketedBinder true >> unicodeSymbol " → " " -> "
def depArrowPrefix   := depArrowShortPrefix <|> depArrowLongPrefix
@[builtinTermParser] def depArrow := leading_parser depArrowPrefix >> termParser
```
Note that no changes in the elaborator are needed.
We decided to not use it because terms such as `{α} → α → α` may look too cryptic.
Note that we did not add a `explicitShortBinder` parser since `(α) → α → α` is really cryptic as a short for `(α : Type) → α → α`.
-/
@[builtinTermParser] def depArrow := leading_parser:25 bracketedBinder true >> unicodeSymbol " → " " -> " >> termParser

def simpleBinder := leading_parser many1 binderIdent >> optType
@[builtinTermParser]
def «forall» := leading_parser:leadPrec unicodeSymbol "∀" "forall" >> many1 (ppSpace >> (simpleBinder <|> bracketedBinder)) >> ", " >> 

def matchAlt (rhsParser : Parser := termParser) : Parser :=
  nodeWithAntiquot "matchAlt" `Lean.Parser.Term.matchAlt $
```

```
  "| " >> ppIndent (sepBy1 termParser ", " >> darrow >> checkColGe "alternative right-hand-side to start in a column greater than or
/--
  Useful for syntax quotations. Note that generic patterns such as `` `(matchAltExpr| | ... => $rhs) `` should also
  work with other `rhsParser`s (of arity 1). -/
def matchAltExpr := matchAlt

def matchAlts (rhsParser : Parser := termParser) : Parser :=
  leading_parser ppDedent $ withPosition $ many1Indent (ppLine >> matchAlt rhsParser)

def matchDiscr := leading_parser optional (atomic (ident >> checkNoWsBefore "no space before ':'" >> ":")) >> termParser

def trueVal  := leading_parser nonReservedSymbol "true"
def falseVal := leading_parser nonReservedSymbol "false"
def generalizingParam := leading_parser atomic ("(" >> nonReservedSymbol "generalizing") >> " := " >> (trueVal <|> falseVal)  >> ")"

@[builtinTermParser] def «match» := leading_parser:leadPrec "match " >> optional generalizingParam >> sepBy1 matchDiscr ", " >> optType
@[builtinTermParser] def «nomatch» := leading_parser:leadPrec "nomatch " >> termParser

def funImplicitBinder := atomic (lookahead ("{" >> many1 binderIdent >> (symbol " : " <|> "}"))) >> implicitBinder
def funSimpleBinder   := atomic (lookahead (many1 binderIdent >> " : ")) >> simpleBinder
def funBinder : Parser := funImplicitBinder <|> instBinder <|> funSimpleBinder <|> termParser maxPrec
-- NOTE: we use `nodeWithAntiquot` to ensure that `fun $b => ...` remains a `term` antiquotation
def basicFun : Parser := nodeWithAntiquot "basicFun" `Lean.Parser.Term.basicFun (many1 (ppSpace >> funBinder) >> darrow >> termParser)
@[builtinTermParser] def «fun» := leading_parser:maxPrec unicodeSymbol "λ" "fun" >> (basicFun <|> matchAlts)

def optExprPrecedence := optional (atomic ":" >> termParser maxPrec)
@[builtinTermParser] def «leading_parser»  := leading_parser:leadPrec "leading_parser " >> optExprPrecedence >> termParser
@[builtinTermParser] def «trailing_parser» := leading_parser:leadPrec "trailing_parser " >> optExprPrecedence >> optExprPrecedence >> t

@[builtinTermParser] def borrowed   := leading_parser "@&" >> termParser leadPrec
@[builtinTermParser] def quotedName := leading_parser nameLit
@[builtinTermParser] def doubleQuotedName := leading_parser "`" >> checkNoWsBefore >> nameLit

def simpleBinderWithoutType := nodeWithAntiquot "simpleBinder" `Lean.Parser.Term.simpleBinder (anonymous := true)
  (many1 binderIdent >> pushNone)

/- Remark: we use `checkWsBefore` to ensure `let x[i] := e; b` is not parsed as `let x [i] := e; b` where `[i]` is an `instBinder`. -/
def letIdLhs    : Parser := ident >> checkWsBefore "expected space before binders" >> many (ppSpace >> (simpleBinderWithoutType <|> bra
def letIdDecl   := nodeWithAntiquot "letIdDecl"   `Lean.Parser.Term.letIdDecl   $ atomic (letIdLhs >> " := ") >> termParser
def letPatDecl  := nodeWithAntiquot "letPatDecl"  `Lean.Parser.Term.letPatDecl  $ atomic (termParser >> pushNone >> optType >> " := ")
def letEqnsDecl := nodeWithAntiquot "letEqnsDecl" `Lean.Parser.Term.letEqnsDecl $ letIdLhs >> matchAlts
-- Remark: we use `nodeWithAntiquot` here to make sure anonymous antiquotations (e.g., `$x`) are not `letDecl`
def letDecl     := nodeWithAntiquot "letDecl" `Lean.Parser.Term.letDecl (notFollowedBy (nonReservedSymbol "rec") "rec" >> (letIdDecl <|
@[builtinTermParser] def «let» := leading_parser:leadPrec  withPosition ("let " >> letDecl) >> optSemicolon termParser
@[builtinTermParser] def «let_fun»     := leading_parser:leadPrec withPosition ((symbol "let_fun " <|> "let_λ") >> letDecl) >> optSemic
@[builtinTermParser] def «let_delayed» := leading_parser:leadPrec withPosition ("let_delayed " >> letDecl) >> optSemicolon termParser
```

```
def «scoped» := leading_parser "scoped "
def «local»  := leading_parser "local "
def attrKind := leading_parser optional («scoped» <|> «local»)
def attrInstance     := ppGroup $ leading_parser attrKind >> attrParser

def attributes       := leading_parser "@[" >> sepBy1 attrInstance ", " >> "]"
def letRecDecl       := leading_parser optional Command.docComment >> optional «attributes» >> letDecl
def letRecDecls      := leading_parser sepBy1 letRecDecl ", "
@[builtinTermParser]
def «letrec» := leading_parser:leadPrec withPosition (group ("let " >> nonReservedSymbol "rec ") >> letRecDecls) >> optSemicolon termPa

@[runBuiltinParserAttributeHooks]
def whereDecls := leading_parser "where " >> many1Indent (group (letRecDecl >> optional ";"))
@[runBuiltinParserAttributeHooks]
def matchAltsWhereDecls := leading_parser matchAlts >> optional whereDecls

@[builtinTermParser] def noindex := leading_parser "no_index " >> termParser maxPrec

@[builtinTermParser] def binrel  := leading_parser "binrel% " >> ident >> ppSpace >> termParser maxPrec >> termParser maxPrec

@[builtinTermParser] def forInMacro := leading_parser "forIn% " >> termParser maxPrec >> termParser maxPrec >> termParser maxPrec

@[builtinTermParser] def typeOf             := leading_parser "typeOf% " >> termParser maxPrec
@[builtinTermParser] def ensureTypeOf       := leading_parser "ensureTypeOf% " >> termParser maxPrec >> strLit >> termParser
@[builtinTermParser] def ensureExpectedType := leading_parser "ensureExpectedType% " >> strLit >> termParser maxPrec
@[builtinTermParser] def noImplicitLambda   := leading_parser "noImplicitLambda% " >> termParser maxPrec

def namedArgument  := leading_parser atomic ("(" >> ident >> " := ") >> termParser >> ")"
def ellipsis       := leading_parser ".."
def argument       :=
  checkWsBefore "expected space" >>
  checkColGt "expected to be indented" >>
  (namedArgument <|> ellipsis <|> termParser argPrec)
-- `app` precedence is `lead` (cannot be used as argument)
-- `lhs` precedence is `max` (i.e. does not accept `arg` precedence)
-- argument precedence is `arg` (i.e. does not accept `lead` precedence)
@[builtinTermParser] def app      := trailing_parser:leadPrec:maxPrec many1 argument

@[builtinTermParser] def proj     := trailing_parser checkNoWsBefore >> "." >> checkNoWsBefore >> (fieldIdx <|> ident)
@[builtinTermParser] def completion := trailing_parser checkNoWsBefore >> "."
@[builtinTermParser] def arrayRef := trailing_parser checkNoWsBefore >> "[" >> termParser >>"]"
@[builtinTermParser] def arrow    := trailing_parser checkPrec 25 >> unicodeSymbol " → " " -> " >> termParser 25

def isIdent (stx : Syntax) : Bool :=
  -- antiquotations should also be allowed where an identifier is expected
  stx.isAntiquot || stx.isIdent
```

```
@[builtinTermParser] def explicitUniv : TrailingParser := trailing_parser checkStackTop isIdent "expected preceding identifier" >> chec
@[builtinTermParser] def namedPattern : TrailingParser := trailing_parser checkStackTop isIdent "expected preceding identifier" >> chec

@[builtinTermParser] def pipeProj   := trailing_parser:minPrec " |>." >> checkNoWsBefore >> (fieldIdx <|> ident) >> many argument
@[builtinTermParser] def pipeCompletion := trailing_parser:minPrec " |>."

@[builtinTermParser] def subst := trailing_parser:75 " ▸ " >> sepBy1 (termParser 75) " ▸ "

-- NOTE: Doesn't call `categoryParser` directly in contrast to most other "static" quotations, so call `evalInsideQuot` explicitly
@[builtinTermParser] def funBinder.quot : Parser := leading_parser "`(funBinder|"  >> toggleInsideQuot (evalInsideQuot ``funBinder funB
def bracketedBinderF := bracketedBinder  -- no default arg
@[builtinTermParser] def bracketedBinder.quot : Parser := leading_parser "`(bracketedBinder|"  >> toggleInsideQuot (evalInsideQuot ``br
@[builtinTermParser] def matchDiscr.quot : Parser := leading_parser "`(matchDiscr|"  >> toggleInsideQuot (evalInsideQuot ``matchDiscr m
@[builtinTermParser] def attr.quot : Parser := leading_parser "`(attr|"  >> toggleInsideQuot attrParser >> ")"

@[builtinTermParser] def panic       := leading_parser:leadPrec "panic! " >> termParser
@[builtinTermParser] def unreachable := leading_parser:leadPrec "unreachable!"
@[builtinTermParser] def dbgTrace    := leading_parser:leadPrec withPosition ("dbg_trace" >> ((interpolatedStr termParser) <|> termPars
@[builtinTermParser] def assert      := leading_parser:leadPrec withPosition ("assert! " >> termParser) >> optSemicolon termParser


def macroArg       := termParser maxPrec
def macroDollarArg := leading_parser "$" >> termParser 10
def macroLastArg   := macroDollarArg <|> macroArg

-- Macro for avoiding exponentially big terms when using `STWorld`
@[builtinTermParser] def stateRefT   := leading_parser "StateRefT" >> macroArg >> macroLastArg

@[builtinTermParser] def dynamicQuot := leading_parser "`(" >> ident >> "|" >> toggleInsideQuot (parserOfStack 1) >> ")"

end Term

@[builtinTermParser default+1] def Tactic.quot    : Parser := leading_parser "`(tactic|" >> toggleInsideQuot tacticParser >> ")"
@[builtinTermParser] def Tactic.quotSeq : Parser := leading_parser "`(tactic|" >> toggleInsideQuot Tactic.seq1 >> ")"

@[builtinTermParser] def Level.quot  : Parser := leading_parser "`(level|" >> toggleInsideQuot levelParser >> ")"

builtin_initialize
  register_parser_alias "letDecl"         Term.letDecl
  register_parser_alias "haveDecl"        Term.haveDecl
  register_parser_alias "sufficesDecl"    Term.sufficesDecl
  register_parser_alias "letRecDecls"     Term.letRecDecls
  register_parser_alias "hole"            Term.hole
  register_parser_alias "syntheticHole"   Term.syntheticHole
  register_parser_alias "matchDiscr"      Term.matchDiscr
  register_parser_alias "bracketedBinder" Term.bracketedBinder
  register_parser_alias "attrKind"        Term.attrKind
```

```

end Parser
end Lean
```