

```

1 ::::::::::::::
2 Elab.lean
3 ::::::::::::::
4 /-
5 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6 Released under Apache 2.0 license as described in the file LICENSE.
7 Authors: Leonardo de Moura
8 -/
9 import Lean.Elab.Import
10 import Lean.Elab.Exception
11 import Lean.Elab.Command
12 import Lean.Elab.Term
13 import Lean.Elab.App
14 import Lean.Elab.Binders
15 import Lean.Elab.LetRec
16 import Lean.Elab.Frontend
17 import Lean.Elab.BuiltinNotation
18 import Lean.Elab.Declaration
19 import Lean.Elab.Tactic
20 import Lean.Elab.Match
21 -- HACK: must come after `Match` because builtin elaborators (for `match` in this case) do not take priorities
22 import Lean.Elab.Quotation
23 import Lean.Elab.Syntax
24 import Lean.Elab.Do
25 import Lean.Elab.StructInst
26 import Lean.Elab.Inductive
27 import Lean.Elab.Structure
28 import Lean.Elab.Print
29 import Lean.Elab.MutualDef
30 import Lean.Elab.PreDefinition
31 import Lean.Elab.Deriving
32 import Lean.Elab.DeclarationRange
33 import Lean.Elab.Extra
34 ::::::::::::::
35 Elab/App.lean
36 ::::::::::::::
37 /-
38 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
39 Released under Apache 2.0 license as described in the file LICENSE.
40 Authors: Leonardo de Moura
41 -/
42 import Lean.Util.FindMVar
43 import Lean.Parser.Term
44 import Lean.Elab.Term
45 import Lean.Elab.Binders
46 import Lean.Elab.SyntheticMVars

```

```

47
48 namespace Lean.Elab.Term
49 open Meta
50
51 builtin_initialize elabWithoutExpectedTypeAttr : TagAttribute ←
52   registerTagAttribute `elabWithoutExpectedType "mark that applications of the given declaration should be elaborated without the expect"
53
54 def hasElabWithoutExpectedType (env : Environment) (declName : Name) : Bool :=
55   elabWithoutExpectedTypeAttr.hasTag env declName
56
57 /--
58   Auxiliary inductive datatype for combining unelaborated syntax
59   and already elaborated expressions. It is used to elaborate applications. -/
60 inductive Arg where
61   | stx (val : Syntax)
62   | expr (val : Expr)
63   deriving Inhabited
64
65 instance : ToString Arg := {fun
66   | Arg.stx val => toString val
67   | Arg.expr val => toString val}
68
69 /-- Named arguments created using the notation `(x := val)` -/
70 structure NamedArg where
71   ref : Syntax := Syntax.missing
72   name : Name
73   val : Arg
74   deriving Inhabited
75
76 instance : ToString NamedArg where
77   toString s := "(" ++ toString s.name ++ " := " ++ toString s.val ++ ")"
78
79 def throwInvalidNamedArg {α} (namedArg : NamedArg) (fn? : Option Name) : TermElabM α :=
80   withRef namedArg.ref <| match fn? with
81     | some fn => throwError "invalid argument name '{namedArg.name}' for function '{fn}'"
82     | none    => throwError "invalid argument name '{namedArg.name}' for function"
83
84 /--
85   Add a new named argument to `namedArgs`, and throw an error if it already contains a named argument
86   with the same name. -/
87 def addNamedArg (namedArgs : Array NamedArg) (namedArg : NamedArg) : TermElabM (Array NamedArg) := do
88   if namedArgs.any (namedArg.name == ·.name) then
89     throwError "argument '{namedArg.name}' was already set"
90   return namedArgs.push namedArg
91
92 private def ensureArgType (f : Expr) (arg : Expr) (expectedType : Expr) : TermElabM Expr := do
93   let argType ← inferType arg

```

```

94 ensureHasTypeAux expectedType argType arg f
95
96 /-
97   Relevant definitions:
98   ```
99   class CoeFun (α : Sort u) (γ : α → outParam (Sort v))
100   abbrev coeFun {α : Sort u} {γ : α → Sort v} (a : α) [CoeFun α γ] : γ a
101   ```
102 -/
103 private def tryCoeFun? (α : Expr) (a : Expr) : TermElabM (Option Expr) := do
104   let v ← mkFreshLevelMVar
105   let type ← mkArrow α (mkSort v)
106   let γ ← mkFreshExprMVar type
107   let u ← getLevel α
108   let coeFunInstType := mkAppN (Lean.mkConst ``CoeFun [u, v]) #[α, γ]
109   let mvar ← mkFreshExprMVar coeFunInstType MetavarKind.synthetic
110   let mvarId := mvar.mvarId!
111   try
112     if (← synthesizeCoeInstMVarCore mvarId) then
113       expandCoe <| mkAppN (Lean.mkConst ``coeFun [u, v]) #[α, γ, a, mvar]
114     else
115       return none
116   catch _ =>
117     return none
118
119 def synthesizeAppInstMVars (instMVars : Array MVarId) : TermElabM Unit :=
120   for mvarId in instMVars do
121     unless (← synthesizeInstMVarCore mvarId) do
122       registerSyntheticMVarWithCurrRef mvarId SyntheticMVarKind.typeClass
123
124 namespace ElabAppArgs
125
126 /- Auxiliary structure for elaborating the application `f args namedArgs`. -/
127 structure State where
128   explicit      : Bool -- true if `@` modifier was used
129   f             : Expr
130   fType         : Expr
131   args          : List Arg           -- remaining regular arguments
132   namedArgs     : List NamedArg     -- remaining named arguments to be processed
133   ellipsis      : Bool := false
134   expectedType? : Option Expr
135   etaArgs       : Array Expr := #[]
136   toSetErrorCtx : Array MVarId := #[] -- metavariables that we need the set the error context using the application being built
137   instMVars     : Array MVarId := #[] -- metavariables for the instance implicit arguments that have already been processed
138   -- The following field is used to implement the `propagateExpectedType` heuristic.
139   propagateExpected : Bool -- true when expectedType has not been propagated yet
140

```

```

141 abbrev M := StateRefT State TermElabM
142
143 /- Add the given metavariable to the collection of metavariables associated with instance-implicit arguments. -/
144 private def addInstMVar (mvarId : MVarId) : M Unit :=
145   modify fun s => { s with instMVars := s.instMVars.push mvarId }
146
147 /-
148   Try to synthesize metavariables are `instMVars` using type class resolution.
149   The ones that cannot be synthesized yet are registered.
150   Remark: we use this method before trying to apply coercions to function. -/
151 def synthesizeAppInstMVars : M Unit := do
152   let s ← get
153   let instMVars := s.instMVars
154   modify fun s => { s with instMVars := #[] }
155   Lean.Elab.Term.synthesizeAppInstMVars instMVars
156
157 /- fType may become a forallE after we synthesize pending metavariables. -/
158 private def synthesizePendingAndNormalizeFunType : M Unit := do
159   synthesizeAppInstMVars
160   synthesizeSyntheticMVars
161   let s ← get
162   let fType ← whnfForall s.fType
163   if fType.isForall then
164     modify fun s => { s with fType := fType }
165   else
166     match (← tryCoeFun? fType s.f) with
167     | some f =>
168       let fType ← inferType f
169       modify fun s => { s with f := f, fType := fType }
170     | none =>
171       for namedArg in s.namedArgs do
172         let f := s.f.getAppFn
173         if f.isConst then
174           throwInvalidNamedArg namedArg f.constName!
175         else
176           throwInvalidNamedArg namedArg none
177       throwError "function expected at{indentExpr s.f}\nterm has type{indentExpr fType}"
178
179 /- Normalize and return the function type. -/
180 private def normalizeFunType : M Expr := do
181   let s ← get
182   let fType ← whnfForall s.fType
183   modify fun s => { s with fType := fType }
184   pure fType
185
186 /- Return the binder name at `fType`. This method assumes `fType` is a function type. -/
187 private def getBindingName : M Name := return (← get).fType.bindingName!

```

```

188
189 /- Return the next argument expected type. This method assumes `fType` is a function type. -/
190 private def getArgExpectedType : M Expr := return (← get).fType.bindingDomain!
191
192 def eraseNamedArgCore (namedArgs : List NamedArg) (binderName : Name) : List NamedArg :=
193   namedArgs.filter (·.name != binderName)
194
195 /- Remove named argument with name `binderName` from `namedArgs`. -/
196 def eraseNamedArg (binderName : Name) : M Unit :=
197   modify fun s => { s with namedArgs := eraseNamedArgCore s.namedArgs binderName }
198
199 /-
200   Add a new argument to the result. That is, `f := f arg`, update `fType`.
201   This method assumes `fType` is a function type. -/
202 private def addNewArg (arg : Expr) : M Unit :=
203   modify fun s => { s with f := mkApp s.f arg, fType := s.fType.bindingBody!.instantiate1 arg }
204
205 /-
206   Elaborate the given `Arg` and add it to the result. See `addNewArg`.
207   Recall that, `Arg` may be wrapping an already elaborated `Expr`. -/
208 private def elabAndAddNewArg (arg : Arg) : M Unit := do
209   let s ← get
210   let expectedType ← getArgExpectedType
211   match arg with
212   | Arg.expr val =>
213     let arg ← ensureArgType s.f val expectedType
214     addNewArg arg
215   | Arg.stx val =>
216     let val ← elabTerm val expectedType
217     let arg ← ensureArgType s.f val expectedType
218     addNewArg arg
219
220 /- Return true if the given type contains `OptParam` or `AutoParams` -/
221 private def hasOptAutoParams (type : Expr) : M Bool := do
222   forallTelescopeReducing type fun xs type =>
223     xs.anyM fun x => do
224       let xType ← inferType x
225       return xType.getOptParamDefault?.isSome || xType.getAutoParamTactic?.isSome
226
227 /- Return true if `fType` contains `OptParam` or `AutoParams` -/
228 private def fTypeHasOptAutoParams : M Bool := do
229   hasOptAutoParams (← get).fType
230
231 /- Auxiliary function for retrieving the resulting type of a function application.
232   See `propagateExpectedType`.
233
234   Remark: `(explicit : Bool) == true` when `@` modifier is used. -/

```

```

235 private partial def getForallBody (explicit : Bool) : Nat → List NamedArg → Expr → Option Expr
236 | i, namedArgs, type@(Expr.forallE n d b c) =>
237   match namedArgs.find? fun (namedArg : NamedArg) => namedArg.name == n with
238   | some _ => getForallBody explicit i (eraseNamedArgCore namedArgs n) b
239   | none =>
240     if !explicit && !c.binderInfo.isExplicit then
241       getForallBody explicit i namedArgs b
242     else if i > 0 then
243       getForallBody explicit (i-1) namedArgs b
244     else if d.isAutoParam || d.isOptParam then
245       getForallBody explicit i namedArgs b
246     else
247       some type
248 | 0, [], type => some type
249 | _, _, _ => none
250
251 private def shouldPropagateExpectedTypeFor (nextArg : Arg) : Bool :=
252   match nextArg with
253   | Arg.expr _ => false -- it has already been elaborated
254   | Arg.stx stx =>
255     -- TODO: make this configurable?
256     stx.getKind != ``Lean.Parser.Term.hole &&
257     stx.getKind != ``Lean.Parser.Term.syntheticHole &&
258     stx.getKind != ``Lean.Parser.Term.byTactic
259
260 /-
261   Auxiliary method for propagating the expected type. We call it as soon as we find the first explicit
262   argument. The goal is to propagate the expected type in applications of functions such as
263   ```lean
264   Add.add {α : Type u} : α → α → α
265   List.cons {α : Type u} : α → List α → List α
266   ```
267   This is particularly useful when there applicable coercions. For example,
268   assume we have a coercion from `Nat` to `Int`, and we have
269   `(x : Nat)` and the expected type is `List Int`. Then, if we don't use this function,
270   the elaborator will fail to elaborate
271   ```
272   List.cons x []
273   ```
274   First, the elaborator creates a new metavariable `?α` for the implicit argument `{α : Type u}`.
275   Then, when it processes `x`, it assigns `?α := Nat`, and then obtain the
276   resultant type `List Nat` which is not definitionally equal to `List Int`.
277   We solve the problem by executing this method before we elaborate the first explicit argument (`x` in this example).
278   This method infers that the resultant type is `List ?α` and unifies it with `List Int`.
279   Then, when we elaborate `x`, the elaborator realizes the coercion from `Nat` to `Int` must be used, and the
280   term
281   ```

```

```

282 @List.cons Int (coe x) (@List.nil Int)
283 ``
284 is produced.
285
286 The method will do nothing if
287 1- The resultant type depends on the remaining arguments (i.e., `!eTypeBody.hasLooseBVars`).
288 2- The resultant type contains optional/auto params.
289
290 We have considered adding the following extra conditions
291 a) The resultant type does not contain any type metavariable.
292 b) The resultant type contains a nontype metavariable.
293
294 These two conditions would restrict the method to simple functions that are "morally" in
295 the Hindley&Milner fragment.
296 If users need to disable expected type propagation, we can add an attribute `[elabWithoutExpectedType]`.
297 -/
298 private def propagateExpectedType (arg : Arg) : M Unit := do
299   if shouldPropagateExpectedTypeFor arg then
300     let s ← get
301     -- TODO: handle s.etaArgs.size > 0
302     unless !s.etaArgs.isEmpty || !s.propagateExpected do
303       match s.expectedType? with
304       | none           => pure ()
305       | some expectedType =>
306         /- We don't propagate `Prop` because we often use `Prop` as a more general "Bool" (e.g., `if-then-else`).
307            If we propagate `expectedType == Prop` in the following examples, the elaborator would fail
308            ```
309            def f1 (s : Nat × Bool) : Bool := if s.2 then false else true
310
311            def f2 (s : List Bool) : Bool := if s.head! then false else true
312
313            def f3 (s : List Bool) : Bool := if List.head! (s.map not) then false else true
314            ```
315            They would all fail for the same reason. So, let's focus on the first one.
316            We would elaborate `s.2` with `expectedType == Prop`.
317            Before we elaborate `s`, this method would be invoked, and `s.fType` is ` $?α × ?β → ?β$ ` and after
318            propagation we would have ` $?α × Prop → Prop$ '. Then, when we would try to elaborate `s`, and
319            get a type error because ` $?α × Prop$ ` cannot be unified with ` $Nat × Bool$ `
320            Most users would have a hard time trying to understand why these examples failed.
321
322            Here is a possible alternative workarounds. We give up the idea of using `Prop` at `if-then-else`.
323            Drawback: users use `if-then-else` with conditions that are not Decidable.
324            So, users would have to embrace `propDecidable` and `choice`.
325            This may not be that bad since the developers and users don't seem to care about constructivism.
326
327            We currently use a different workaround, we just don't propagate the expected type when it is `Prop`. -/
328     if expectedType.isProp then

```

```

329     modify fun s => { s with propagateExpected := false }
330   else
331     let numRemainingArgs := s.args.length
332     trace[Elab.app.propagateExpectedType] "etaArgs.size: {s.etaArgs.size}, numRemainingArgs: {numRemainingArgs}, fType: {s.fType}"
333     match getForallBody s.explicit numRemainingArgs s.namedArgs s.fType with
334     | none => pure ()
335     | some fTypeBody =>
336       unless fTypeBody.hasLooseBVars do
337         unless (← hasOptAutoParams fTypeBody) do
338           trace[Elab.app.propagateExpectedType] "{expectedType} =?= {fTypeBody}"
339           if (← isDefEq expectedType fTypeBody) then
340             /- Note that we only set `propagateExpected := false` when propagation has succeeded. -/
341             modify fun s => { s with propagateExpected := false }
342
343 /-
344 Create a fresh local variable with the current binder name and argument type, add it to `etaArgs` and `f`,
345 and then execute the continuation `k`.-/
346 private def addEtaArg (k : M Expr) : M Expr := do
347   let n ← getBindingName
348   let type ← getArgExpectedType
349   withLocalDeclD n type fun x => do
350     modify fun s => { s with etaArgs := s.etaArgs.push x }
351     addNewArg x
352     k
353
354 /- This method execute after all application arguments have been processed. -/
355 private def finalize : M Expr := do
356   let s ← get
357   let mut e := s.f
358   -- all user explicit arguments have been consumed
359   trace[Elab.app.finalize] e
360   let ref ← getRef
361   -- Register the error context of implicits
362   for mvarId in s.toSetErrorCtx do
363     registerMVarErrorImplicitArgInfo mvarId ref e
364   if !s.etaArgs.isEmpty then
365     e ← mkLambdaFVars s.etaArgs e
366 /-
367 Remark: we should not use `s.fType` as `eType` even when
368 `s.etaArgs.isEmpty`. Reason: it may have been unfolded.
369 -/
370   let eType ← inferType e
371   trace[Elab.app.finalize] "after etaArgs, {e} : {eType}"
372   match s.expectedType? with
373   | none => pure ()
374   | some expectedType =>
375     trace[Elab.app.finalize] "expected type: {expectedType}"

```



```

376     -- Try to propagate expected type. Ignore if types are not definitionally equal, caller must handle it.
377     discard <| isDefEq expectedType eType
378     synthesizeAppInstMVars
379     pure e
380
381 private def addImplicitArg (k : M Expr) : M Expr := do
382   let argType ← getArgExpectedType
383   let arg ← mkFreshExprMVar argType
384   modify fun s => { s with toSetErrorCtx := s.toSetErrorCtx.push arg.mvarId! }
385   addNewArg arg
386   k
387
388 /- Return true if there is a named argument that depends on the next argument. -/
389 private def anyNamedArgDependsOnCurrent : M Bool := do
390   let s ← get
391   if s.namedArgs.isEmpty then
392     return false
393   else
394     forallTelescopeReducing s.fType fun xs _ => do
395       let curr := xs[0]
396       for i in [1:xs.size] do
397         let xDecl ← getLocalDecl xs[i].fvarId!
398         if s.namedArgs.any fun arg => arg.name == xDecl.userName then
399           if (← getMCtx).localDeclDependsOn xDecl curr.fvarId! then
400             return true
401       return false
402
403 /-
404   Process a `fType` of the form `(x : A) → B x`.
405   This method assume `fType` is a function type -/
406 private def processExplicitArg (k : M Expr) : M Expr := do
407   let s ← get
408   match s.args with
409   | arg::args =>
410     propagateExpectedType arg
411     modify fun s => { s with args := args }
412     elabAndAddNewArg arg
413     k
414   | _ =>
415     let argType ← getArgExpectedType
416     match s.explicit, argType.getOptParamDefault?, argType.getAutoParamTactic? with
417     | false, some defVal, _ => addNewArg defVal; k
418     | false, _, some (Expr.const tacticDecl _ _) =>
419       let env ← getEnv
420       let opts ← getOptions
421       match evalSyntaxConstant env opts tacticDecl with
422       | Except.error err      => throwError err

```

```

423 | Except.ok tacticSyntax =>
424   -- TODO(Leo): does this work correctly for tactic sequences?
425   let tacticBlock ← `(by $tacticSyntax)
426   let argType     := argType.getArg! 0 -- `autoParam type := by tactic` ==> `type`
427   let argNew := Arg.stx tacticBlock
428   propagateExpectedType argNew
429   elabAndAddNewArg argNew
430   k
431 | false, _, some _ =>
432   throwError "invalid autoParam, argument must be a constant"
433 | _, _, _ =>
434   if !s.namedArgs.isEmpty then
435     if (← anyNamedArgDependsOnCurrent) then
436       addImplicitArg k
437     else
438       addEtaArg k
439   else if !s.explicit then
440     if (← fTypeHasOptAutoParams) then
441       addEtaArg k
442     else if (← get).ellipsis then
443       addImplicitArg k
444     else
445       finalize
446   else
447     finalize
448
449 /-
450   Process a `fType` of the form `{x : A} → B x`.
451   This method assume `fType` is a function type -/
452 private def processImplicitArg (k : M Expr) : M Expr := do
453   if (← get).explicit then
454     processExplicitArg k
455   else
456     addImplicitArg k
457
458 /- Return true if the next argument at `args` is of the form ` _ ` -/
459 private def isNextArgHole : M Bool := do
460   match (← get).args with
461   | Arg.stx (Syntax.node ``Lean.Parser.Term.hole _) :: _ => pure true
462   | _ => pure false
463
464 /-
465   Process a `fType` of the form `[x : A] → B x`.
466   This method assume `fType` is a function type -/
467 private def processInstImplicitArg (k : M Expr) : M Expr := do
468   if (← get).explicit then
469     if (← isNextArgHole) then

```

```

470     /- Recall that if '@' has been used, and the argument is '_', then we still use type class resolution -/
471     let arg ← mkFreshExprMVar (← getArgExpectedType) MetavarKind.synthetic
472     modify fun s => { s with args := s.args.tail! }
473     addInstMVar arg.mvarId!
474     addNewArg arg
475     k
476   else
477     processExplictArg k
478 else
479   let arg ← mkFreshExprMVar (← getArgExpectedType) MetavarKind.synthetic
480   addInstMVar arg.mvarId!
481   addNewArg arg
482   k
483
484 /- Return true if there are regular or named arguments to be processed. -/
485 private def hasArgsToProcess : M Bool := do
486   let s ← get
487   pure $ !s.args.isEmpty || !s.namedArgs.isEmpty
488
489 /- Elaborate function application arguments. -/
490 partial def main : M Expr := do
491   let s ← get
492   let fType ← normalizeFunType
493   if fType.isForall then
494     let binderName := fType.bindingName!
495     let binfo := fType.bindingInfo!
496     let s ← get
497     match s.namedArgs.find? fun (namedArg : NamedArg) => namedArg.name == binderName with
498     | some namedArg =>
499       propagateExpectedType namedArg.val
500       eraseNamedArg binderName
501       elabAndAddNewArg namedArg.val
502       main
503     | none =>
504       match binfo with
505       | BinderInfo.implicit      => processImplicitArg main
506       | BinderInfo.instImplicit => processInstImplicitArg main
507       | _                       => processExplictArg main
508   else if (← hasArgsToProcess) then
509     synthesizePendingAndNormalizeFunType
510     main
511   else
512     finalize
513
514 end ElabAppArgs
515
516 private def propagateExpectedTypeFor (f : Expr) : TermElabM Bool :=

```

```

517 match f.getAppFn.constName? with
518 | some declName => return !hasElabWithoutExpectedType (← getEnv) declName
519 | _ => return true
520
521 def elabAppArgs (f : Expr) (namedArgs : Array NamedArg) (args : Array Arg)
522   (expectedType? : Option Expr) (explicit ellipsis : Bool) : TermElabM Expr := do
523   let fType ← inferType f
524   let fType ← instantiateMVars fType
525   trace[Elab.app.args] "explicit: {explicit}, {f} : {fType}"
526   unless namedArgs.isEmpty && args.isEmpty do
527     tryPostponeIfMVar fType
528   ElabAppArgs.main.run' {
529     args := args.toList,
530     expectedType? := expectedType?,
531     explicit := explicit,
532     ellipsis := ellipsis,
533     namedArgs := namedArgs.toList,
534     f := f,
535     fType := fType
536     propagateExpected := (← propagateExpectedTypeFor f)
537   }
538
539 /- Auxiliary inductive datatype that represents the resolution of an `LVal`. -/
540 inductive LValResolution where
541 | projFn (baseStructName : Name) (structName : Name) (fieldName : Name)
542 | projIdx (structName : Name) (idx : Nat)
543 | const (baseStructName : Name) (structName : Name) (constName : Name)
544 | localRec (baseName : Name) (fullName : Name) (fvar : Expr)
545 | getOp (fullName : Name) (idx : Syntax)
546
547 private def throwLValError {α} (e : Expr) (eType : Expr) (msg : MessageData) : TermElabM α :=
548   throwError "{msg}{indentExpr e}\nhas type{indentExpr eType}"
549
550 /- `findMethod? env S fName`.
551   1- If `env` contains `S ++ fName`, return `(S, S++fName)`
552   2- Otherwise if `env` contains private name `prv` for `S ++ fName`, return `(S, prv)`, o
553   3- Otherwise for each parent structure `S` of `S`, we try `findMethod? env S' fName` -/
554 private partial def findMethod? (env : Environment) (structName fieldName : Name) : Option (Name × Name) :=
555   let fullName := structName ++ fieldName
556   match env.find? fullName with
557   | some _ => some (structName, fullName)
558   | none =>
559     let fullNamePrv := mkPrivateName env fullName
560     match env.find? fullNamePrv with
561     | some _ => some (structName, fullNamePrv)
562     | none =>
563       if isStructureLike env structName then

```

```

564     (getParentStructures env structName).findSome? fun parentStructName => findMethod? env parentStructName fieldName
565   else
566     none
567
568 private def resolveLValAux (e : Expr) (eType : Expr) (lval : LVal) : TermElabM LValResolution := do
569   match eType.getAppFn.constName?, lval with
570   | some structName, LVal.fieldIdx _ idx =>
571     if idx == 0 then
572       throwError "invalid projection, index must be greater than 0"
573     let env ← getEnv
574     unless isStructureLike env structName do
575       throwLValError e eType "invalid projection, structure expected"
576     let fieldNames := getStructureFields env structName
577     if h : idx - 1 < fieldNames.size then
578       if isStructure env structName then
579         return LValResolution.projFn structName structName (fieldNames.get (idx - 1, h))
580       else
581         /- `structName` was declared using `inductive` command.
582         So, we don't projection functions for it. Thus, we use `Expr.proj` -/
583         return LValResolution.projIdx structName (idx - 1)
584   else
585     throwLValError e eType m!"invalid projection, structure has only {fieldNames.size} field(s)"
586 | some structName, LVal.fieldName _ fieldName _ =>
587   let env ← getEnv
588   let searchEnv : Unit → TermElabM LValResolution := fun _ => do
589     match findMethod? env structName (Name.mkSimple fieldName) with
590     | some (baseStructName, fullName) => pure $ LValResolution.const baseStructName structName fullName
591     | none =>
592       throwLValError e eType
593       m!"invalid field '{fieldName}', the environment does not contain '{Name.mkStr structName fieldName}'"
594   -- search local context first, then environment
595   let searchCtx : Unit → TermElabM LValResolution := fun _ => do
596     let fullName := Name.mkStr structName fieldName
597     let currNamespace ← getCurrNamespace
598     let localName := fullName.replacePrefix currNamespace Name.anonymous
599     let lctx ← getLCtx
600     match lctx.findFromUserName? localName with
601     | some localDecl =>
602       if localDecl.binderInfo == BinderInfo.auxDecl then
603         /- LVal notation is being used to make a "local" recursive call. -/
604         pure $ LValResolution.localRec structName fullName localDecl.toExpr
605       else
606         searchEnv ()
607     | none => searchEnv ()
608   if isStructure env structName then
609     match findField? env structName (Name.mkSimple fieldName) with
610     | some baseStructName => pure $ LValResolution.projFn baseStructName structName (Name.mkSimple fieldName)

```

```

611 | none => searchCtx ()
612 else
613   searchCtx ()
614 | some structName, LVal.getOp _ idx =>
615   let env ← getEnv
616   let fullName := Name.mkStr structName "getOp"
617   match env.find? fullName with
618   | some _ => pure $ LValResolution.getOp fullName idx
619   | none => throwLValError e eType m!"invalid [...] notation because environment does not contain '{fullName}'"
620 | none, LVal.fieldName _ _ (some suffix) _ =>
621   if e.isConst then
622     throwUnknownConstant (e.constName! ++ suffix)
623   else
624     throwLValError e eType "invalid field notation, type is not of the form (C ...) where C is a constant"
625 | _, LVal.getOp _ idx =>
626   throwLValError e eType "invalid [...] notation, type is not of the form (C ...) where C is a constant"
627 | _, _ =>
628   throwLValError e eType "invalid field notation, type is not of the form (C ...) where C is a constant"
629
630 /- whnfCore + implicit consumption.
631 Example: given `e` with `eType := {α : Type} → (fun β => List β) α`, it produces `(e ?m, List ?m)` where `?m` is fresh metavariable.
632 private partial def consumeImplicits (stx : Syntax) (e eType : Expr) : TermElabM (Expr × Expr) := do
633   let eType ← whnfCore eType
634   match eType with
635   | Expr.forallE n d b c =>
636     if c.binderInfo.isImplicit then
637       let mvar ← mkFreshExprMVar d
638       registerMVarErrorHoleInfo mvar.mvarId! stx
639       consumeImplicits stx (mkApp e mvar) (b.instantiate1 mvar)
640     else if c.binderInfo.isInstImplicit then
641       let mvar ← mkInstMVar d
642       consumeImplicits stx (mkApp e mvar) (b.instantiate1 mvar)
643     else match d.getOptParamDefault? with
644     | some defVal => consumeImplicits stx (mkApp e defVal) (b.instantiate1 defVal)
645     -- TODO: we do not handle autoParams here.
646     | _ => pure (e, eType)
647 | _ => pure (e, eType)
648
649 private partial def resolveLValLoop (lval : LVal) (e eType : Expr) (previousExceptions : Array Exception) : TermElabM (Expr × LValResolu
650 let (e, eType) ← consumeImplicits lval.getRef e eType
651 tryPostponeIfMVar eType
652 try
653   let lvalRes ← resolveLValAux e eType lval
654   pure (e, lvalRes)
655 catch
656 | ex@(Exception.error _ _) =>
657   let eType? ← unfoldDefinition? eType

```

```

658   match eType? with
659   | some eType => resolveLValLoop lval e eType (previousExceptions.push ex)
660   | none      =>
661     previousExceptions.forM fun ex => logException ex
662     throw ex
663   | ex@(Exception.internal _ _) => throw ex
664
665 private def resolveLVal (e : Expr) (lval : LVal) : TermElabM (Expr × LValResolution) := do
666   let eType ← inferType e
667   resolveLValLoop lval e eType #[]
668
669 private partial def mkBaseProjections (baseStructName : Name) (structName : Name) (e : Expr) : TermElabM Expr := do
670   let env ← getEnv
671   match getPathToBaseStructure? env baseStructName structName with
672   | none => throwError "failed to access field in parent structure"
673   | some path =>
674     let mut e := e
675     for projFunName in path do
676       let projFn ← mkConst projFunName
677       e ← elabAppArgs projFn #[{ name := `self, val := Arg.expr e }] (args := #[]) (expectedType? := none) (explicit := false) (ellipsis := false)
678     return e
679
680 /- Auxiliary method for field notation. It tries to add `e` as a new argument to `args` or `namedArgs`.
681 This method first finds the parameter with a type of the form `(baseName ...)`.
682 When the parameter is found, if it an explicit one and `args` is big enough, we add `e` to `args`.
683 Otherwise, if there isn't another parameter with the same name, we add `e` to `namedArgs`.
684
685 Remark: `fullName` is the name of the resolved "field" access function. It is used for reporting errors -/
686 private def addLValArg (baseName : Name) (fullName : Name) (e : Expr) (args : Array Arg) (namedArgs : Array NamedArg) (fType : Expr)
687   : TermElabM (Array Arg × Array NamedArg) :=
688   forallTelescopeReducing fType fun xs _ => do
689     let mut argIdx := 0 -- position of the next explicit argument
690     let mut remainingNamedArgs := namedArgs
691     for i in [:xs.size] do
692       let x := xs[i]
693       let xDecl ← getLocalDecl x.fvarId!
694       /- If there is named argument with name `xDecl.userName`, then we skip it. -/
695       match remainingNamedArgs.findIdx? (fun namedArg => namedArg.name == xDecl.userName) with
696       | some idx =>
697         remainingNamedArgs := remainingNamedArgs.eraseIdx idx
698       | none =>
699         let mut foundIt := false
700         let type := xDecl.type
701         if type.consumeMData.isAppOf baseName then
702           foundIt := true
703         if !foundIt then
704           /- Normalize type and try again -/

```

```

705     let type ← withReducible $ whnf type
706     if type.consumeMData.isAppOf baseName then
707       foundIt := true
708   if foundIt then
709     /- We found a type of the form (baseName ...).
710       First, we check if the current argument is an explicit one,
711       and the current explicit position "fits" at `args` (i.e., it must be ≤ arg.size) -/
712     if argIdx ≤ args.size && xDecl.binderInfo.isExplicit then
713       /- We insert `e` as an explicit argument -/
714       return (args.insertAt argIdx (Arg.expr e), namedArgs)
715     /- If we can't add `e` to `args`, we try to add it using a named argument, but this is only possible
716       if there isn't an argument with the same name occurring before it. -/
717     for j in [:i] do
718       let prev := xs[j]
719       let prevDecl ← getLocalDecl prev.fvarId!
720       if prevDecl.userName == xDecl.userName then
721         throwError "invalid field notation, function '{fullName}' has argument with the expected type{indentExpr type}\nbut it can
722         return (args, namedArgs.push { name := xDecl.userName, val := Arg.expr e })
723     if xDecl.binderInfo.isExplicit then
724       -- advance explicit argument position
725       argIdx := argIdx + 1
726   throwError "invalid field notation, function '{fullName}' does not have argument with type ({baseName} ...) that can be used, it mus
727
728 private def elabAppLValsAux (namedArgs : Array NamedArg) (args : Array Arg) (expectedType? : Option Expr) (explicit ellipsis : Bool)
729   (f : Expr) (lvals : List LVal) : TermElabM Expr :=
730   let rec loop : Expr → List LVal → TermElabM Expr
731   | f, [] => elabAppArgs f namedArgs args expectedType? explicit ellipsis
732   | f, lval::lvals => do
733     if let LVal.fieldName (ref := fieldStx) (targetStx := targetStx) .. := lval then
734       addDotCompletionInfo targetStx f expectedType? fieldStx
735       let (f, lvalRes) ← resolveLVal f lval
736       match lvalRes with
737       | LValResolution.projIdx structName idx =>
738         let f := mkProj structName idx f
739         addTermInfo lval.getRef f
740         loop f lvals
741       | LValResolution.projFn baseStructName structName fieldName =>
742         let f ← mkBaseProjections baseStructName structName f
743         let projFn ← mkConst (baseStructName ++ fieldName)
744         addTermInfo lval.getRef projFn
745         if lvals.isEmpty then
746           let namedArgs ← addNamedArg namedArgs { name := `self, val := Arg.expr f }
747           elabAppArgs projFn namedArgs args expectedType? explicit ellipsis
748         else
749           let f ← elabAppArgs projFn #[{ name := `self, val := Arg.expr f }] #[] (expectedType? := none) (explicit := false) (ellipsis :=
750           loop f lvals
751   | LValResolution.const baseStructName structName constName =>

```



```

752   let f ← if baseStructName != structName then mkBaseProjections baseStructName structName f else pure f
753   let projFn ← mkConst constName
754   addTermInfo lval.getRef projFn
755   if lvals.isEmpty then
756     let projFnType ← inferType projFn
757     let (args, namedArgs) ← addLValArg baseStructName constName f args namedArgs projFnType
758     elabAppArgs projFn namedArgs args expectedType? explicit ellipsis
759   else
760     let f ← elabAppArgs projFn #[] #[Arg.expr f] (expectedType? := none) (explicit := false) (ellipsis := false)
761     loop f lvals
762 | LValResolution.localRec baseName fullName fvar =>
763   addTermInfo lval.getRef fvar
764   if lvals.isEmpty then
765     let fvarType ← inferType fvar
766     let (args, namedArgs) ← addLValArg baseName fullName f args namedArgs fvarType
767     elabAppArgs fvar namedArgs args expectedType? explicit ellipsis
768   else
769     let f ← elabAppArgs fvar #[] #[Arg.expr f] (expectedType? := none) (explicit := false) (ellipsis := false)
770     loop f lvals
771 | LValResolution.getOp fullName idx =>
772   let getOpFn ← mkConst fullName
773   addTermInfo lval.getRef getOpFn
774   if lvals.isEmpty then
775     let namedArgs ← addNamedArg namedArgs { name := `self, val := Arg.expr f }
776     let namedArgs ← addNamedArg namedArgs { name := `idx, val := Arg.stx idx }
777     elabAppArgs getOpFn namedArgs args expectedType? explicit ellipsis
778   else
779     let f ← elabAppArgs getOpFn #[{ name := `self, val := Arg.expr f }, { name := `idx, val := Arg.stx idx }]
780     #[] (expectedType? := none) (explicit := false) (ellipsis := false)
781   loop f lvals
782 loop f lvals
783
784 private def elabAppLVals (f : Expr) (lvals : List LVal) (namedArgs : Array NamedArg) (args : Array Arg)
785   (expectedType? : Option Expr) (explicit ellipsis : Bool) : TermElabM Expr := do
786   if !lvals.isEmpty && explicit then
787     throwError "invalid use of field notation with `@` modifier"
788   elabAppLValsAux namedArgs args expectedType? explicit ellipsis f lvals
789
790 def elabExplicitUnivs (lvls : Array Syntax) : TermElabM (List Level) := do
791   lvls.foldrM (fun stx lvls => do pure ((← elabLevel stx)::lvls)) []
792
793 /-
794 Interaction between `errToSorry` and `observing`.
795
796 - The method `elabTerm` catches exceptions, log them, and returns a synthetic sorry (IF `ctx.errToSorry` == true).
797
798 - When we elaborate choice nodes (and overloaded identifiers), we track multiple results using the `observing x` combinator.

```

```

799   The `observing x` executes `x` and returns a `TermElabResult`.
800
801   `observing x` does not check for synthetic sorry's, just an exception. Thus, it may think `x` worked when it didn't
802   if a synthetic sorry was introduced. We decided that checking for synthetic sorrys at `observing` is not a good solution
803   because it would not be clear to decide what the "main" error message for the alternative is. When the result contains
804   a synthetic `sorry`, it is not clear which error message corresponds to the `sorry`. Moreover, while executing `x`, many
805   error messages may have been logged. Recall that we need an error per alternative at `mergeFailures`.
806
807   Thus, we decided to set `errToSorry` to `false` whenever processing choice nodes and overloaded symbols.
808
809   Important: we rely on the property that after `errToSorry` is set to
810   false, no elaboration function executed by `x` will reset it to
811   `true`.
812   -/
813
814 private partial def elabAppFnId (fIdent : Syntax) (fExplicitUnivs : List Level) (lvals : List LVal)
815   (namedArgs : Array NamedArg) (args : Array Arg) (expectedType? : Option Expr) (explicit ellipsis overloaded : Bool) (acc : Array (Te
816   : TermElabM (Array (TermElabResult Expr)) := do
817   let funLVals ← withRef fIdent <| resolveName' fIdent fExplicitUnivs expectedType?
818   let overloaded := overloaded || funLVals.length > 1
819   -- Set `errToSorry` to `false` if `funLVals` > 1. See comment above about the interaction between `errToSorry` and `observing`.
820   withReader (fun ctx => { ctx with errToSorry := funLVals.length == 1 && ctx.errToSorry }) do
821     funLVals.foldLM (init := acc) fun acc (f, fIdent, fields) => do
822       addTermInfo fIdent f
823       let lvals' := toLVals fields (first := true)
824       let s ← observing do
825         let e ← elabAppLVals f (lvals' ++ lvals) namedArgs args expectedType? explicit ellipsis
826         if overloaded then ensureHasType expectedType? e else pure e
827       return acc.push s
828 where
829   toName : List Syntax → Name
830   | [] => Name.anonymous
831   | field :: fields => Name.mkStr (toName fields) field.getId.toString
832
833   toLVals : List Syntax → (first : Bool) → List LVal
834   | [], _ => []
835   | field :: fields, true => LVal.fieldName field field.getId.toString (toName (field :: fields)) fIdent :: toLVals fields false
836   | field :: fields, false => LVal.fieldName field field.getId.toString none fIdent :: toLVals fields false
837
838 private partial def elabAppFn (f : Syntax) (lvals : List LVal) (namedArgs : Array NamedArg) (args : Array Arg)
839   (expectedType? : Option Expr) (explicit ellipsis overloaded : Bool) (acc : Array (TermElabResult Expr)) : TermElabM (Array (TermElab
840   if f.getKind == choiceKind then
841     -- Set `errToSorry` to `false` when processing choice nodes. See comment above about the interaction between `errToSorry` and `obser
842     withReader (fun ctx => { ctx with errToSorry := false }) do
843       f.getArgs.foldLM (fun acc f => elabAppFn f lvals namedArgs args expectedType? explicit ellipsis true acc) acc
844   else
845     let elabFieldName (e field : Syntax) := do

```

```

846   let newLVals := field.getId.eraseMacroScopes.components.map fun n =>
847     -- We use `none` here since `field` can't be part of a composite name
848     LVal.fieldName field (toString n) none e
849   elabAppFn e (newLVals ++ lvals) namedArgs args expectedType? explicit ellipsis overloaded acc
850 let elabFieldIdx (e idxStx : Syntax) := do
851   let idx := idxStx.isFieldIdx?.get!
852   elabAppFn e (LVal.fieldIdx idxStx idx :: lvals) namedArgs args expectedType? explicit ellipsis overloaded acc
853 match f with
854 | `($e).$idx:fieldIdx => elabFieldIdx e idx
855 | `($e |>.$idx:fieldIdx) => elabFieldIdx e idx
856 | `($e).$field:ident => elabFieldName e field
857 | `($e |>.$field:ident) => elabFieldName e field
858 | `($e[%$bracket $idx]) => elabAppFn e (LVal.getOp bracket idx :: lvals) namedArgs args expectedType? explicit ellipsis overloaded a
859 | `($id:ident@$t:term) =>
860   throwError "unexpected occurrence of named pattern"
861 | `($id:ident) => do
862   elabAppFnId id [] lvals namedArgs args expectedType? explicit ellipsis overloaded acc
863 | `($id:ident.{sus,*}) => do
864   let us ← elabExplicitUnivs us
865   elabAppFnId id us lvals namedArgs args expectedType? explicit ellipsis overloaded acc
866 | `(@$id:ident) =>
867   elabAppFn id lvals namedArgs args expectedType? (explicit := true) ellipsis overloaded acc
868 | `(@$id:ident.{sus,*}) =>
869   elabAppFn (f.getArg 1) lvals namedArgs args expectedType? (explicit := true) ellipsis overloaded acc
870 | `(@$t)      => throwUnsupportedSyntax -- invalid occurrence of `@`
871 | `(_)       => throwError "placeholders '_' cannot be used where a function is expected"
872 | _          => do
873   let catchPostpone := !overloaded
874   /- If we are processing a choice node, then we should use `catchPostpone == false` when elaborating terms.
875   Recall that `observing` does not catch `postponeExceptionId`. -/
876   if lvals.isEmpty && namedArgs.isEmpty && args.isEmpty then
877     /- Recall that elabAppFn is used for elaborating atomics terms **and** choice nodes that may contain
878     arbitrary terms. If they are not being used as a function, we should elaborate using the expectedType. -/
879     let s ←
880       if overloaded then
881         observing <| elabTermEnsuringType f expectedType? catchPostpone
882       else
883         observing <| elabTerm f expectedType?
884     return acc.push s
885   else
886     let s ← observing do
887       let f ← elabTerm f none catchPostpone
888       let e ← elabAppLVals f lvals namedArgs args expectedType? explicit ellipsis
889       if overloaded then ensureHasType expectedType? e else pure e
890     return acc.push s
891
892 private def isSuccess (candidate : TermElabResult Expr) : Bool :=

```

```

893 match candidate with
894 | EStateM.Result.ok _ _ => true
895 | _ => false
896
897 private def getSuccess (candidates : Array (TermElabResult Expr)) : Array (TermElabResult Expr) :=
898   candidates.filter isSuccess
899
900 private def toMessageData (ex : Exception) : TermElabM MessageData := do
901   let pos ← getRefPos
902   match ex.getRef.getPos? with
903   | none      => return ex.toMessageData
904   | some xPos =>
905     if pos == xPos then
906       return ex.toMessageData
907     else
908       let exPosition := (← getFileMap).toPosition xPos
909       return m!"{exPosition.line}:{exPosition.column} {ex.toMessageData}"
910
911 private def toMessageList (msgs : Array MessageData) : MessageData :=
912   indentD (MessageData.joinSep msgs.toList m!"\n\n")
913
914 private def mergeFailures {α} (failures : Array (TermElabResult Expr)) : TermElabM α := do
915   let msgs ← failures.mapM fun failure =>
916     match failure with
917     | EStateM.Result.ok _ _      => unreachable!
918     | EStateM.Result.error ex _ => toMessageData ex
919   throwError "overloaded, errors {toMessageList msgs}"
920
921 private def elabAppAux (f : Syntax) (namedArgs : Array NamedArg) (args : Array Arg) (ellipsis : Bool) (expectedType? : Option Expr) : Te
922   let candidates ← elabAppFn f [] namedArgs args expectedType? (explicit := false) (ellipsis := ellipsis) (overloaded := false) #[]
923   if candidates.size == 1 then
924     applyResult candidates[0]
925   else
926     let successes := getSuccess candidates
927     if successes.size == 1 then
928       applyResult successes[0]
929     else if successes.size > 1 then
930       let lctx ← getLCtx
931       let opts ← getOptions
932       let msgs : Array MessageData := successes.map fun success => match success with
933         | EStateM.Result.ok e s => MessageData.withContext { env := s.meta.core.env, mctx := s.meta.meta.mctx, lctx := lctx, opts := opt
934         | _                      => unreachable!
935       throwErrorAt f "ambiguous, possible interpretations {toMessageList msgs}"
936     else
937       withRef f <| mergeFailures candidates
938
939 partial def expandArgs (args : Array Syntax) (pattern := false) : TermElabM (Array NamedArg × Array Arg × Bool) := do

```

```

940 let (args, ellipsis) :=
941   if args.isEmpty then
942     (args, false)
943   else if args.back.isOfKind ``Lean.Parser.Term.ellipsis then
944     (args.pop, true)
945   else
946     (args, false)
947 let (namedArgs, args) ← args.foldlM (init := (#[], #[])) fun (namedArgs, args) stx => do
948   if stx.getKind == ``Lean.Parser.Term.namedArgument then
949     -- trailing_tpaser try "(" >> ident >> " := " >> termParser >> ")"
950     let name := stx[1].getId.eraseMacroScopes
951     let val := stx[3]
952     let namedArgs ← addNamedArg namedArgs { ref := stx, name := name, val := Arg.stx val }
953     return (namedArgs, args)
954   else if stx.getKind == ``Lean.Parser.Term.ellipsis then
955     throwErrorAt stx "unexpected '..."
956   else
957     return (namedArgs, args.push $ Arg.stx stx)
958 return (namedArgs, args, ellipsis)
959
960 def expandApp (stx : Syntax) (pattern := false) : TermElabM (Syntax × Array NamedArg × Array Arg × Bool) := do
961   let (namedArgs, args, ellipsis) ← expandArgs stx[1].getArgs
962   return (stx[0], namedArgs, args, ellipsis)
963
964 @[builtinTermElab app] def elabApp : TermElab := fun stx expectedType? =>
965   withoutPostponingUniverseConstraints do
966     let (f, namedArgs, args, ellipsis) ← expandApp stx
967     elabAppAux f namedArgs args (ellipsis := ellipsis) expectedType?
968
969 private def elabAtom : TermElab := fun stx expectedType? =>
970   elabAppAux stx #[] #[] (ellipsis := false) expectedType?
971
972 @[builtinTermElab ident] def elabIdent : TermElab := elabAtom
973 @[builtinTermElab namedPattern] def elabNamedPattern : TermElab := elabAtom
974 @[builtinTermElab explicitUniv] def elabExplicitUniv : TermElab := elabAtom
975 @[builtinTermElab pipeProj] def elabPipeProj : TermElab
976 | `($e |>.$f $args*), expectedType? =>
977   withoutPostponingUniverseConstraints do
978     let (namedArgs, args, ellipsis) ← expandArgs args
979     elabAppAux (← `($e |>.$f)) namedArgs args (ellipsis := ellipsis) expectedType?
980 | _, _ => throwUnsupportedSyntax
981
982 @[builtinTermElab explicit] def elabExplicit : TermElab := fun stx expectedType? =>
983   match stx with
984   | `(@$id:ident) => elabAtom stx expectedType? -- Recall that `elabApp` also has support for `@`
985   | `(@$id:ident.{us,*}) => elabAtom stx expectedType?
986   | `(@($t)) => elabTerm t expectedType? (implicitLambda := false) -- `@` is being used just to disable implicit lambdas

```

```

987 | `(@$t)          => elabTerm t expectedType? (implicitLambda := false)  -- `@` is being used just to disable implicit lambdas
988 | _              => throwUnsupportedSyntax
989
990 @[builtinTermElab choice] def elabChoice : TermElab := elabAtom
991 @[builtinTermElab proj]  def elabProj  : TermElab := elabAtom
992 @[builtinTermElab arrayRef] def elabArrayRef : TermElab := elabAtom
993
994 builtin_initialize
995   registerTraceClass `Elab.app
996
997 end Lean.Elab.Term
998 ::::::::::::::
999 Elab/Attributes.lean
1000 ::::::::::::::
1001 /-
1002 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
1003 Released under Apache 2.0 license as described in the file LICENSE.
1004 Authors: Leonardo de Moura, Sebastian Ullrich
1005 -/
1006 import Lean.Parser.Attr
1007 import Lean.Attributes
1008 import Lean.MonadEnv
1009 import Lean.Elab.Util
1010 namespace Lean.Elab
1011
1012 structure Attribute where
1013   kind : AttributeKind := AttributeKind.global
1014   name  : Name
1015   stx   : Syntax := Syntax.missing
1016
1017 instance : ToFormat Attribute where
1018   format attr :=
1019     let kindStr := match attr.kind with
1020     | AttributeKind.global => ""
1021     | AttributeKind.local  => "local "
1022     | AttributeKind.scoped => "scoped "
1023     Format.bracket "@" f!"{kindStr}{attr.name}{toString attr.stx}" "]"
1024
1025 instance : Inhabited Attribute where
1026   default := { name := arbitrary }
1027
1028 /-
1029   ``
1030   attrKind := leading_parser optional («scoped» <|> «local»)
1031   ``
1032 -/
1033 def toAttributeKind [Monad m] [MonadResolveName m] [MonadError m] (attrKindStx : Syntax) : m AttributeKind := do

```

```

1034 if attrKindStx[0].isNone then
1035   return AttributeKind.global
1036 else if attrKindStx[0][0].getKind == ``Lean.Parser.Term.scoped then
1037   if (← getCurrNamespace).isAnonymous then
1038     throwError "scoped attributes must be used inside namespaces"
1039   return AttributeKind.scoped
1040 else
1041   return AttributeKind.local
1042
1043 def mkAttrKindGlobal : Syntax :=
1044   Syntax.node ``Lean.Parser.Term.attrKind #[mkNullNode]
1045
1046 def elabAttr {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (attrInstance : Synt
1047   /- attrInstance      := ppGroup $ leading_parser attrKind >> attrParser -/
1048   let attrKind ← toAttributeKind attrInstance[0]
1049   let attr := attrInstance[1]
1050   let attr ← liftMacroM <| expandMacros attr
1051   let attrName ←
1052     if attr.getKind == ``Parser.Attr.simple then
1053       pure attr[0].getId.eraseMacroScopes
1054     else
1055       match attr.getKind with
1056       | Name.str _ s _ => pure <| Name.mkSimple s
1057       | _ => throwErrorAt attr "unknown attribute"
1058   unless isAttribute (← getEnv) attrName do
1059     throwError "unknown attribute [{attrName}]"
1060   /- The `AttrM` does not have sufficient information for expanding macros in `args`.
1061       So, we expand them before here before we invoke the attributer handlers implemented using `AttrM`. -/
1062   pure { kind := attrKind, name := attrName, stx := attr }
1063
1064 def elabAttrs {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (attrInstances : Ar
1065   let mut attrs := #[]
1066   for attr in attrInstances do
1067     attrs := attrs.push (← elabAttr attr)
1068   return attrs
1069
1070 -- leading_parser "@[" >> sepBy1 attrInstance ", " >> "]"
1071 def elabDeclAttrs {m} [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m] (stx : Syntax)
1072   elabAttrs stx[1].getSepArgs
1073
1074 end Lean.Elab
1075 ::::::::::::::
1076 Elab/AutoBound.lean
1077 ::::::::::::::
1078 /-
1079 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
1080 Released under Apache 2.0 license as described in the file LICENSE.

```

```

1081 Authors: Leonardo de Moura
1082 -/
1083 import Lean.Data.Options
1084
1085 /- Basic support for auto bound implicit local names -/
1086
1087 namespace Lean.Elab
1088
1089 register_builtin_option autoBoundImplicitLocal : Bool := {
1090   defValue := true
1091   descr    := "Unbound local variables in declaration headers become implicit arguments if they are a lower case or greek letter follo
1092 }
1093
1094 private def isValidAutoBoundSuffix (s : String) : Bool :=
1095   s.toSubstring.drop 1 |>.all fun c => c.isDigit || isSubScriptAlnum c || c == '_' || c == '\\'
1096
1097 /-
1098 Remark: Issue #255 exposed a nasty interaction between macro scopes and auto-bound-implicit names.
1099 ```
1100 local notation "A" => id x
1101 theorem test : A = A := sorry
1102 ```
1103 We used to use `n.eraseMacroScopes` at `isValidAutoBoundImplicitName` and `isValidAutoBoundLevelName`.
1104 Thus, in the example above, when `A` is expanded, a `x` with a fresh macro scope is created.
1105 `x`+macro-scope is not in scope and is a valid auto-bound implicit name after macro scopes are erased.
1106 So, an auto-bound exception would be thrown, and `x`+macro-scope would be added as a new implicit.
1107 When, we try again, a `x` with a new macro scope is created and this process keeps repeating.
1108 Therefore, we do consider identifier with macro scopes anymore.
1109 -/
1110
1111 def isValidAutoBoundImplicitName (n : Name) : Bool :=
1112   match n with
1113   | Name.str Name.anonymous s _ => s.length > 0 && (isGreek s[0] || s[0].isLower) && isValidAutoBoundSuffix s
1114   | _ => false
1115
1116 def isValidAutoBoundLevelName (n : Name) : Bool :=
1117   match n with
1118   | Name.str Name.anonymous s _ => s.length > 0 && s[0].isLower && isValidAutoBoundSuffix s
1119   | _ => false
1120
1121 end Lean.Elab
1122 :::::::::::::::
1123 Elab/Binders.lean
1124 :::::::::::::::
1125 /-
1126 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
1127 Released under Apache 2.0 license as described in the file LICENSE.

```



```

1128 Authors: Leonardo de Moura
1129 -/
1130 import Lean.Elab.Term
1131 import Lean.Parser.Term
1132
1133 namespace Lean.Elab.Term
1134 open Meta
1135 open Lean.Parser.Term
1136
1137 /--
1138   Given syntax of the forms
1139     a) `(` `term)?
1140     b) `(` `term
1141   return `term` if it is present, or a hole if not. -/
1142 private def expandBinderType (ref : Syntax) (stx : Syntax) : Syntax :=
1143   if stx.getNumArgs == 0 then
1144     mkHole ref
1145   else
1146     stx[1]
1147
1148 /-- Given syntax of the form `ident <|> hole`, return `ident`. If `hole`, then we create a new anonymous name. -/
1149 private def expandBinderIdent (stx : Syntax) : TermElabM Syntax :=
1150   match stx with
1151   | `(_) => mkFreshIdent stx
1152   | _    => pure stx
1153
1154 /-- Given syntax of the form `(ident >> " : ")?`, return `ident`, or a new instance name. -/
1155 private def expandOptIdent (stx : Syntax) : TermElabM Syntax := do
1156   if stx.isNone then
1157     let id ← withFreshMacroScope <| MonadQuotation.addMacroScope `inst
1158     return mkIdentFrom stx id
1159   else
1160     return stx[0]
1161
1162 structure BinderView where
1163   id : Syntax
1164   type : Syntax
1165   bi : BinderInfo
1166
1167 partial def quoteAutoTactic : Syntax → TermElabM Syntax
1168 | stx@(Syntax.ident _ _ _) => throwErrorAt stx "invalid auto tactic, identifier is not allowed"
1169 | stx@(Syntax.node k args) => do
1170   if stx.isAntiquot then
1171     throwErrorAt stx "invalid auto tactic, antiquotation is not allowed"
1172   else
1173     let mut quotedArgs ← `(Array.empty)
1174     for arg in args do

```

```

1175     if k == nullKind && (arg.isAntiquotSuffixSplice || arg.isAntiquotSplice) then
1176       throwErrorAt arg "invalid auto tactic, antiquotation is not allowed"
1177     else
1178       let quotedArg ← quoteAutoTactic arg
1179       quotedArgs ← `(Array.push $quotedArgs $quotedArg)
1180       `(Syntax.node $(quote k) $quotedArgs)
1181   | Syntax.atom info val => `(mkAtom $(quote val))
1182   | Syntax.missing      => unreachable!
1183
1184 def declareTacticSyntax (tactic : Syntax) : TermElabM Name :=
1185   withFreshMacroScope do
1186     let name ← MonadQuotation.addMacroScope `_auto
1187     let type := Lean.mkConst `Lean.Syntax
1188     let tactic ← quoteAutoTactic tactic
1189     let val ← elabTerm tactic type
1190     let val ← instantiateMVars val
1191     trace[Elab.autoParam] val
1192     let decl := Declaration.defnDecl { name := name, levelParams := [], type := type, value := val, hints := ReducibilityHints.opaque,
1193                                     safety := DefinitionSafety.safe }
1194     addDecl decl
1195     compileDecl decl
1196     return name
1197
1198 /-
1199 Expand `optional (binderTactic <|> binderDefault)`
1200 def binderTactic := leading_parser " := " >> " by " >> tacticParser
1201 def binderDefault := leading_parser " := " >> termParser
1202 -/
1203 private def expandBinderModifier (type : Syntax) (optBinderModifier : Syntax) : TermElabM Syntax := do
1204   if optBinderModifier.isNone then
1205     return type
1206   else
1207     let modifier := optBinderModifier[0]
1208     let kind := modifier.getKind
1209     if kind == `Lean.Parser.Term.binderDefault then
1210       let defaultVal := modifier[1]
1211       `(optParam $type $defaultVal)
1212     else if kind == `Lean.Parser.Term.binderTactic then
1213       let tac := modifier[2]
1214       let name ← declareTacticSyntax tac
1215       `(autoParam $type $(mkIdentFrom tac name))
1216     else
1217       throwUnsupportedSyntax
1218
1219 private def getBinderIds (ids : Syntax) : TermElabM (Array Syntax) :=
1220   ids.getArgs.mapM fun id =>
1221     let k := id.getKind

```

```

1222     if k == identKind || k == `Lean.Parser.Term.hole then
1223         return id
1224     else
1225         throwErrorAt id "identifier or `_' expected"
1226
1227 /-
1228   Recall that
1229   ```
1230   def typeSpec := leading_parser " : " >> termParser
1231   def optType : Parser := optional typeSpec
1232   ```
1233 -/
1234 def expandOptType (ref : Syntax) (optType : Syntax) : Syntax :=
1235     if optType.isNone then
1236         mkHole ref
1237     else
1238         optType[0][1]
1239
1240 private def matchBinder (stx : Syntax) : TermElabM (Array BinderView) := do
1241     let k := stx.getKind
1242     if k == `Lean.Parser.Term.simpleBinder then
1243         -- binderIdent+ >> optType
1244         let ids ← getBinderIds stx[0]
1245         let type := expandOptType stx stx[1]
1246         ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.default }
1247     else if k == `Lean.Parser.Term.explicitBinder then
1248         -- `(` binderIdent+ binderType (binderDefault <|> binderTactic)? `)`
1249         let ids ← getBinderIds stx[1]
1250         let type := expandBinderType stx stx[2]
1251         let optModifier := stx[3]
1252         let type ← expandBinderModifier type optModifier
1253         ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.default }
1254     else if k == `Lean.Parser.Term.implicitBinder then
1255         -- `{` binderIdent+ binderType `}`
1256         let ids ← getBinderIds stx[1]
1257         let type := expandBinderType stx stx[2]
1258         ids.mapM fun id => do pure { id := (← expandBinderIdent id), type := type, bi := BinderInfo.implicit }
1259     else if k == `Lean.Parser.Term.instBinder then
1260         -- `[` optIdent type `]`
1261         let id ← expandOptIdent stx[1]
1262         let type := stx[2]
1263         pure #[ { id := id, type := type, bi := BinderInfo.instImplicit } ]
1264     else
1265         throwUnsupportedSyntax
1266
1267 private def registerFailedToInferBinderTypeInfo (type : Expr) (ref : Syntax) : TermElabM Unit :=
1268     registerCustomErrorIfMVar type ref "failed to infer binder type"

```

```

1269
1270 private def addLocalVarInfoCore (lctx : LocalContext) (stx : Syntax) (fvar : Expr) : TermElabM Unit := do
1271   if (← getInfoState).enabled then
1272     pushInfoTree <| InfoTree.node (children := {}) <| Info.ofTermInfo { lctx := lctx, expr := fvar, stx := stx }
1273
1274 private def addLocalVarInfo (stx : Syntax) (fvar : Expr) : TermElabM Unit := do
1275   addLocalVarInfoCore (← getLCtx) stx fvar
1276
1277 private def ensureAtomicBinderName (binderView : BinderView) : TermElabM Unit :=
1278   let n := binderView.id.getId.eraseMacroScopes
1279   unless n.isAtomic do
1280     throwErrorAt binderView.id "invalid binder name '{n}', it must be atomic"
1281
1282 private partial def elabBinderViews {α} (binderViews : Array BinderView) (fvvars : Array Expr) (k : Array Expr → TermElabM α)
1283   : TermElabM α :=
1284   let rec loop (i : Nat) (fvvars : Array Expr) : TermElabM α := do
1285     if h : i < binderViews.size then
1286       let binderView := binderViews.get (i, h)
1287       ensureAtomicBinderName binderView
1288       let type ← elabType binderView.type
1289       registerFailedToInferBinderTypeInfo type binderView.type
1290       withLocalDecl binderView.id.getId binderView.bi type fun fvar => do
1291         addLocalVarInfo binderView.id fvar
1292         loop (i+1) (fvvars.push fvar)
1293     else
1294       k fvvars
1295   loop 0 fvvars
1296
1297 private partial def elabBindersAux {α} (binders : Array Syntax) (k : Array Expr → TermElabM α) : TermElabM α :=
1298   let rec loop (i : Nat) (fvvars : Array Expr) : TermElabM α := do
1299     if h : i < binders.size then
1300       let binderViews ← matchBinder (binders.get (i, h))
1301       elabBinderViews binderViews fvvars <| loop (i+1)
1302     else
1303       k fvvars
1304   loop 0 #[]
1305
1306 /--
1307   Elaborate the given binders (i.e., `Syntax` objects for `simpleBinder <|> bracketedBinder`),
1308   update the local context, set of local instances, reset instance cache (if needed), and then
1309   execute `x` with the updated context. -/
1310 def elabBinders {α} (binders : Array Syntax) (k : Array Expr → TermElabM α) : TermElabM α :=
1311   withoutPostponingUniverseConstraints do
1312     if binders.isEmpty then
1313       k #[]
1314     else
1315       elabBindersAux binders k

```

```

1316
1317 @[inline] def elabBinder {α} (binder : Syntax) (x : Expr → TermElabM α) : TermElabM α :=
1318   elabBinders #[binder] fun fvars => x fvars[0]
1319
1320 @[builtinTermElab «forall»] def elabForall : TermElab := fun stx _ =>
1321   match stx with
1322   | `(forall $binders*, $term) =>
1323     elabBinders binders fun xs => do
1324       let e ← elabType term
1325       mkForallFVars xs e
1326   | _ => throwUnsupportedSyntax
1327
1328 @[builtinTermElab arrow] def elabArrow : TermElab :=
1329   adaptExpander fun stx => match stx with
1330   | `($dom:term -> $rng) => `(forall (a : $dom), $rng)
1331   | _ => throwUnsupportedSyntax
1332
1333 @[builtinTermElab depArrow] def elabDepArrow : TermElab := fun stx _ =>
1334   -- bracketedBinder `->` term
1335   let binder := stx[0]
1336   let term   := stx[2]
1337   elabBinders #[binder] fun xs => do
1338     mkForallFVars xs (← elabType term)
1339
1340 /--
1341   Auxiliary functions for converting `id_1 ... id_n` application into `[id_1, ..., id_m]`
1342   It is used at `expandFunBinders`. -/
1343 private partial def getFunBinderIds? (stx : Syntax) : OptionT TermElabM (Array Syntax) :=
1344   let convertElem (stx : Syntax) : OptionT TermElabM Syntax :=
1345     match stx with
1346     | `(_) => do let ident ← mkFreshIdent stx; pure ident
1347     | `($id:ident) => return id
1348     | _ => failure
1349   match stx with
1350   | `($f $args*) => do
1351     let mut acc := #[].push (← convertElem f)
1352     for arg in args do
1353       acc := acc.push (← convertElem arg)
1354     return acc
1355   | _ =>
1356     return #[].push (← convertElem stx)
1357
1358 /--
1359   Auxiliary function for expanding `fun` notation binders. Recall that `fun` parser is defined as
1360   ```
1361   def funBinder : Parser := implicitBinder <|> instBinder <|> termParser maxPrec
1362   leading_parser unicodeSymbol "λ" "fun" >> many1 funBinder >> "=>" >> termParser

```

```

1363   ``
1364   to allow notation such as `fun (a, b) => a + b`, where `(a, b)` should be treated as a pattern.
1365   The result is a pair `(explicitBinders, newBody)`, where `explicitBinders` is syntax of the form
1366   ``
1367   `(` ident `:` term `)`
1368   ``
1369   which can be elaborated using `elabBinders`, and `newBody` is the updated `body` syntax.
1370   We update the `body` syntax when expanding the pattern notation.
1371   Example: `fun (a, b) => a + b` expands into `fun _a_1 => match _a_1 with | (a, b) => a + b`.
1372   See local function `processAsPattern` at `expandFunBindersAux`.
1373
1374   The resulting `Bool` is true if a pattern was found. We use it "mark" a macro expansion. -/
1375 partial def expandFunBinders (binders : Array Syntax) (body : Syntax) : TermElabM (Array Syntax × Syntax × Bool) :=
1376   let rec loop (body : Syntax) (i : Nat) (newBinders : Array Syntax) := do
1377     if h : i < binders.size then
1378       let binder := binders.get (i, h)
1379       let processAsPattern : Unit → TermElabM (Array Syntax × Syntax × Bool) := fun _ => do
1380         let pattern := binder
1381         let major ← mkFreshIdent binder
1382         let (binders, newBody, _) ← loop body (i+1) (newBinders.push $ mkExplicitBinder major (mkHole binder))
1383         let newBody ← `(match $major:ident with | $pattern => $newBody)
1384         pure (binders, newBody, true)
1385       match binder with
1386       | Syntax.node `Lean.Parser.Term.implicitBinder _ => loop body (i+1) (newBinders.push binder)
1387       | Syntax.node `Lean.Parser.Term.instBinder _ => loop body (i+1) (newBinders.push binder)
1388       | Syntax.node `Lean.Parser.Term.explicitBinder _ => loop body (i+1) (newBinders.push binder)
1389       | Syntax.node `Lean.Parser.Term.simpleBinder _ => loop body (i+1) (newBinders.push binder)
1390       | Syntax.node `Lean.Parser.Term.hole _ =>
1391         let ident ← mkFreshIdent binder
1392         let type := binder
1393         loop body (i+1) (newBinders.push <| mkExplicitBinder ident type)
1394       | Syntax.node `Lean.Parser.Term.paren args =>
1395         -- `(` (termParser >> parenSpecial)? `)`
1396         -- parenSpecial := (tupleTail <|> typeAscription)?
1397         let binderBody := binder[1]
1398         if binderBody.isNone then
1399           processAsPattern ()
1400         else
1401           let idents := binderBody[0]
1402           let special := binderBody[1]
1403           if special.isNone then
1404             processAsPattern ()
1405           else if special[0].getKind != `Lean.Parser.Term.typeAscription then
1406             processAsPattern ()
1407           else
1408             -- typeAscription := `:` term
1409             let type := special[0][1]

```

```

1410         match (← getFunBinderIds? ident) with
1411         | some ident => loop body (i+1) (newBinders ++ ident.map (fun ident => mkExplicitBinder ident type))
1412         | none      => processAsPattern ()
1413     | Syntax.ident .. =>
1414         let type := mkHole binder
1415         loop body (i+1) (newBinders.push <| mkExplicitBinder binder type)
1416     | _ => processAsPattern ()
1417 else
1418     pure (newBinders, body, false)
1419 loop body 0 #[]
1420
1421 namespace FunBinders
1422
1423 structure State where
1424   fvars      : Array Expr := #[]
1425   lctx       : LocalContext
1426   localInsts : LocalInstances
1427   expectedType? : Option Expr := none
1428
1429 private def propagateExpectedType (fvar : Expr) (fvarType : Expr) (s : State) : TermElabM State := do
1430   match s.expectedType? with
1431   | none      => pure s
1432   | some expectedType =>
1433       let expectedType ← whnfForall expectedType
1434       match expectedType with
1435       | Expr.forallE _ d b _ =>
1436           discard <| isDefEq fvarType d
1437           let b := b.instantiate1 fvar
1438           pure { s with expectedType? := some b }
1439       | _ => pure { s with expectedType? := none }
1440
1441 private partial def elabFunBinderViews (binderViews : Array BinderView) (i : Nat) (s : State) : TermElabM State := do
1442   if h : i < binderViews.size then
1443       let binderView := binderViews.get (i, h)
1444       ensureAtomicBinderName binderView
1445       withRef binderView.type <| withLCtx s.lctx s.localInsts do
1446           let type ← elabType binderView.type
1447           registerFailedToInferBinderTypeInfo type binderView.type
1448           let fvarId ← mkFreshFVarId
1449           let fvar := mkFVar fvarId
1450           let s := { s with fvars := s.fvars.push fvar }
1451           -- dbgTrace (toString binderView.id.getId ++ " : " ++ toString type)
1452           /-
1453           We do not want to support default and auto arguments in lambda abstractions.
1454           Example: `fun (x : Nat := 10) => x+1`.
1455           We do not believe this is an useful feature, and it would complicate the logic here.
1456           -/

```

```

1457     let lctx := s.lctx.mkLocalDecl fvarId binderView.id.getId type binderView.bi
1458     addLocalVarInfoCore lctx binderView.id fvar
1459     let s ← withRef binderView.id <| propagateExpectedType fvar type s
1460     let s := { s with lctx := lctx }
1461     match (← isClass? type) with
1462     | none      => elabFunBinderViews binderViews (i+1) s
1463     | some className =>
1464         resettingSynthInstanceCache do
1465             let localInsts := s.localInsts.push { className := className, fvar := mkFVar fvarId }
1466             elabFunBinderViews binderViews (i+1) { s with localInsts := localInsts }
1467     else
1468         pure s
1469
1470 partial def elabFunBindersAux (binders : Array Syntax) (i : Nat) (s : State) : TermElabM State := do
1471     if h : i < binders.size then
1472         let binderViews ← matchBinder (binders.get (i, h))
1473         let s ← elabFunBinderViews binderViews 0 s
1474         elabFunBindersAux binders (i+1) s
1475     else
1476         pure s
1477
1478 end FunBinders
1479
1480 def elabFunBinders {α} (binders : Array Syntax) (expectedType? : Option Expr) (x : Array Expr → Option Expr → TermElabM α) : TermElabM α
1481 if binders.isEmpty then
1482     x #[] expectedType?
1483 else do
1484     let lctx ← getLCtx
1485     let localInsts ← getLocalInstances
1486     let s ← FunBinders.elabFunBindersAux binders 0 { lctx := lctx, localInsts := localInsts, expectedType? := expectedType? }
1487     resettingSynthInstanceCacheWhen (s.localInsts.size > localInsts.size) <| withLCtx s.lctx s.localInsts <|
1488         x s.fvars s.expectedType?
1489
1490 /- Helper function for `expandEqnsIntoMatch` -/
1491 private def getMatchAltsNumPatterns (matchAlts : Syntax) : Nat :=
1492     let alt0 := matchAlts[0][0]
1493     let pats := alt0[1].getSepArgs
1494     pats.size
1495
1496 def expandWhereDecls (whereDecls : Syntax) (body : Syntax) : MacroM Syntax :=
1497     match whereDecls with
1498     | `(whereDecls|where $[decls:letRecDecl $[;]?]*) => `(let rec $decls:letRecDecl,*; $body)
1499     | _ => Macro.throwUnsupported
1500
1501 def expandWhereDeclsOpt (whereDeclsOpt : Syntax) (body : Syntax) : MacroM Syntax :=
1502     if whereDeclsOpt.isNone then
1503         body

```



```

1504 else
1505   expandWhereDecls whereDeclsOpt[0] body
1506
1507 /- Helper function for `expandMatchAltsIntoMatch` -/
1508 private def expandMatchAltsIntoMatchAux (matchAlts : Syntax) (matchTactic : Bool) : Nat → Array Syntax → MacroM Syntax
1509 | 0,   discrs => do
1510   if matchTactic then
1511     `(tactic|match [$discrs:term],* with $matchAlts:matchAlts)
1512   else
1513     `(match [$discrs:term],* with $matchAlts:matchAlts)
1514 | n+1, discrs => withFreshMacroScope do
1515   let x ← `(x)
1516   let d ← `(@$x:ident) -- See comment below
1517   let body ← expandMatchAltsIntoMatchAux matchAlts matchTactic n (discrs.push d)
1518   if matchTactic then
1519     `(tactic| intro $x:term; $body:tactic)
1520   else
1521     `(@fun $x => $body)
1522
1523 /-
1524   Expand `matchAlts` syntax into a full `match`-expression.
1525   Example
1526   ```
1527   | 0, true => alt_1
1528   | i, _    => alt_2
1529   ```
1530   expands into (for tactic == false)
1531   ```
1532   fun x_1 x_2 =>
1533     match @x_1, @x_2 with
1534     | 0, true => alt_1
1535     | i, _    => alt_2
1536   ```
1537   and (for tactic == true)
1538   ```
1539   intro x_1; intro x_2;
1540   match @x_1, @x_2 with
1541   | 0, true => alt_1
1542   | i, _    => alt_2
1543   ```
1544
1545   Remark: we add `@` to make sure we don't consume implicit arguments, and to make the behavior consistent with `fun`.
1546   Example:
1547   ```
1548   inductive T : Type 1 :=
1549   | mkT : (forall {a : Type}, a -> a) -> T
1550

```

```

1551 def makeT (f : forall {a : Type}, a -> a) : T :=
1552   mkT f
1553
1554 def makeT' : (forall {a : Type}, a -> a) -> T
1555 | f => mkT f
1556 ```
1557 The two definitions should be elaborated without errors and be equivalent.
1558 -/
1559 def expandMatchAltsIntoMatch (ref : Syntax) (matchAlts : Syntax) (tactic := false) : MacroM Syntax :=
1560   withRef ref <| expandMatchAltsIntoMatchAux matchAlts tactic (getMatchAltsNumPatterns matchAlts) #[]
1561
1562 def expandMatchAltsIntoMatchTactic (ref : Syntax) (matchAlts : Syntax) : MacroM Syntax :=
1563   withRef ref <| expandMatchAltsIntoMatchAux matchAlts true (getMatchAltsNumPatterns matchAlts) #[]
1564
1565 /--
1566   Similar to `expandMatchAltsIntoMatch`, but supports an optional `where` clause.
1567
1568   Expand `matchAltsWhereDecls` into `let rec` + `match`-expression.
1569   Example
1570   ```
1571   | 0, true => ... f 0 ...
1572   | i, _    => ... f i + g i ...
1573   where
1574     f x := g x + 1
1575
1576     g : Nat → Nat
1577     | 0    => 1
1578     | x+1 => f x
1579   ```
1580   expands into
1581   ```
1582   fun x_1 x_2 =>
1583     let rec
1584       f x := g x + 1,
1585       g : Nat → Nat
1586       | 0    => 1
1587       | x+1 => f x
1588     match x_1, x_2 with
1589     | 0, true => ... f 0 ...
1590     | i, _    => ... f i + g i ...
1591   ```
1592 -/
1593 def expandMatchAltsWhereDecls (matchAltsWhereDecls : Syntax) : MacroM Syntax :=
1594   let matchAlts      := matchAltsWhereDecls[0]
1595   let whereDeclsOpt := matchAltsWhereDecls[1]
1596   let rec loop (i : Nat) (discrs : Array Syntax) : MacroM Syntax :=
1597     match i with

```

```

1598 | 0 => do
1599   let matchStx ← `(match $[discrs:term],* with $matchAlts:matchAlts)
1600   if whereDeclsOpt.isNone then
1601     return matchStx
1602   else
1603     expandWhereDeclsOpt whereDeclsOpt matchStx
1604 | n+1 => withFreshMacroScope do
1605   let x ← `(x)
1606   let d ← `(@$x:ident) -- See comment at `expandMatchAltsIntoMatch`
1607   let body ← loop n (discrs.push d)
1608   `(@fun $x => $body)
1609 loop (getMatchAltsNumPatterns matchAlts) #[]
1610
1611 @[builtinTermElab «fun»] partial def elabFun : TermElab :=
1612   fun stx expectedType? => loop stx expectedType?
1613 where
1614   loop (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr :=
1615     match stx with
1616     | `(fun $binders* => $body) => do
1617       let (binders, body, expandedPattern) ← expandFunBinders binders body
1618       if expandedPattern then
1619         let newStx ← `(fun $binders* => $body)
1620         loop newStx expectedType?
1621       else
1622         elabFunBinders binders expectedType? fun xs expectedType? => do
1623           /- We ensure the expectedType here since it will force coercions to be applied if needed.
1624             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`,
1625             and another instance for the dependent version. -/
1626           let e ← elabTermEnsuringType body expectedType?
1627           mkLambdaFVars xs e
1628     | `(fun $m:matchAlts) => do
1629       let stxNew ← liftMacroM $ expandMatchAltsIntoMatch stx m
1630       withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?
1631     | _ => throwUnsupportedSyntax
1632
1633 /- If `useLetExpr` is true, then a kernel let-expression `let x : type := val; body` is created.
1634 Otherwise, we create a term of the form `(fun (x : type) => body) val`
1635
1636 The default elaboration order is `binders`, `typeStx`, `valStx`, and `body`.
1637 If `elabBodyFirst == true`, then we use the order `binders`, `typeStx`, `body`, and `valStx`. -/
1638 def elabLetDeclAux (id : Syntax) (binders : Array Syntax) (typeStx : Syntax) (valStx : Syntax) (body : Syntax)
1639   (expectedType? : Option Expr) (useLetExpr : Bool) (elabBodyFirst : Bool) : TermElabM Expr := do
1640   let (type, val, arity) ← elabBinders binders fun xs => do
1641     let type ← elabType typeStx
1642     registerCustomErrorIfMVar type typeStx "failed to infer 'let' declaration type"
1643   if elabBodyFirst then
1644     let type ← mkForallFVars xs type

```

```

1645     let val ← mkFreshExprMVar type
1646     pure (type, val, xs.size)
1647   else
1648     let val ← elabTermEnsuringType valStx type
1649     let type ← mkForallFVars xs type
1650     let val ← mkLambdaFVars xs val
1651     pure (type, val, xs.size)
1652   trace[Elab.let.decl] "{id.getId} : {type} := {val}"
1653   let result ←
1654     if useLetExpr then
1655       withLetDecl id.getId type val fun x => do
1656         addLocalVarInfo id x
1657         let body ← elabTerm body expectedType?
1658         let body ← instantiateMVars body
1659         mkLetFVars #[x] body
1660     else
1661       let f ← withLocalDecl id.getId BinderInfo.default type fun x => do
1662         addLocalVarInfo id x
1663         let body ← elabTerm body expectedType?
1664         let body ← instantiateMVars body
1665         mkLambdaFVars #[x] body
1666       pure <| mkApp f val
1667   if elabBodyFirst then
1668     forallBoundedTelescope type arity fun xs type => do
1669       let valResult ← elabTermEnsuringType valStx type
1670       let valResult ← mkLambdaFVars xs valResult
1671       unless (← isDefEq val valResult) do
1672         throwError "unexpected error when elaborating 'let'"
1673   pure result
1674
1675 structure LetIdDeclView where
1676   id      : Syntax
1677   binders : Array Syntax
1678   type    : Syntax
1679   value   : Syntax
1680
1681 def mkLetIdDeclView (letIdDecl : Syntax) : LetIdDeclView :=
1682   -- `letIdDecl` is of the form `ident >> many bracketedBinder >> optType >> " := " >> termParser
1683   let id      := letIdDecl[0]
1684   let binders := letIdDecl[1].getArgs
1685   let optType := letIdDecl[2]
1686   let type    := expandOptType letIdDecl optType
1687   let value   := letIdDecl[4]
1688   { id := id, binders := binders, type := type, value := value }
1689
1690 def expandLetEqnsDecl (letDecl : Syntax) : MacroM Syntax := do
1691   let ref      := letDecl

```

```

1692 let matchAlts := letDecl[3]
1693 let val ← expandMatchAltsIntoMatch ref matchAlts
1694 return Syntax.node `Lean.Parser.Term.letIdDecl #[letDecl[0], letDecl[1], letDecl[2], mkAtomFrom ref " := ", val]
1695
1696 def elabLetDeclCore (stx : Syntax) (expectedType? : Option Expr) (useLetExpr : Bool) (elabBodyFirst : Bool) : TermElabM Expr := do
1697   let ref      := stx
1698   let letDecl  := stx[1][0]
1699   let body     := stx[3]
1700   if letDecl.getKind == `Lean.Parser.Term.letIdDecl then
1701     let { id := id, binders := binders, type := type, value := val } := mkLetIdDeclView letDecl
1702     elabLetDeclAux id binders type val body expectedType? useLetExpr elabBodyFirst
1703   else if letDecl.getKind == `Lean.Parser.Term.letPatDecl then
1704     -- node `Lean.Parser.Term.letPatDecl $ try (termParser >> pushNone >> optType >> " := ") >> termParser
1705     let pat      := letDecl[0]
1706     let optType  := letDecl[2]
1707     let type     := expandOptType stx optType
1708     let val      := letDecl[4]
1709     let stxNew ← `(let x : $type := $val; match x with | $pat => $body)
1710     let stxNew := match useLetExpr, elabBodyFirst with
1711     | true, false => stxNew
1712     | true, true  => stxNew.setKind `Lean.Parser.Term.«let_delayed»
1713     | false, true  => stxNew.setKind `Lean.Parser.Term.«let_fun»
1714     | false, false => unreachable!
1715     withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
1716   else if letDecl.getKind == `Lean.Parser.Term.letEqnsDecl then
1717     let letDeclIdNew ← liftMacroM <| expandLetEqnsDecl letDecl
1718     let declNew := stx[1].setArg 0 letDeclIdNew
1719     let stxNew := stx.setArg 1 declNew
1720     withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
1721   else
1722     throwUnsupportedSyntax
1723
1724 @[builtinTermElab «let»] def elabLetDecl : TermElab :=
1725   fun stx expectedType? => elabLetDeclCore stx expectedType? true false
1726
1727 @[builtinTermElab «let_fun»] def elabLetFunDecl : TermElab :=
1728   fun stx expectedType? => elabLetDeclCore stx expectedType? false false
1729
1730 @[builtinTermElab «let_delayed»] def elabLetDelayedDecl : TermElab :=
1731   fun stx expectedType? => elabLetDeclCore stx expectedType? true true
1732
1733 builtin_initialize registerTraceClass `Elab.let
1734
1735 end Lean.Elab.Term
1736 ::::::::::::::
1737 Elab/BuiltinNotation.lean
1738 ::::::::::::::

```

```

1739 /-
1740 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
1741 Released under Apache 2.0 license as described in the file LICENSE.
1742 Authors: Leonardo de Moura
1743 -/
1744 import Init.Data.ToString
1745 import Lean.Compiler.BorrowedAnnotation
1746 import Lean.Meta.KAbstract
1747 import Lean.Meta.Transform
1748 import Lean.Elab.Term
1749 import Lean.Elab.SyntheticMVars
1750
1751 namespace Lean.Elab.Term
1752 open Meta
1753
1754 @[builtinTermElab anonymousCtor] def elabAnonymousCtor : TermElab := fun stx expectedType? =>
1755   match stx with
1756   | `(($args,*)) => do
1757     tryPostponeIfNoneOrMVar expectedType?
1758     match expectedType? with
1759     | some expectedType =>
1760       let expectedType ← whnf expectedType
1761       matchConstInduct expectedType.getAppFn
1762       (fun _ => throwError "invalid constructor {...}, expected type must be an inductive type {indentExpr expectedType}")
1763       (fun ival us => do
1764         match ival.ctors with
1765         | [ctor] =>
1766           let cinfo ← getConstInfoCtor ctor
1767           let numExplicitFields ← forallTelescopeReducing cinfo.type fun xs _ => do
1768             let mut n := 0
1769             for i in [cinfo.numParams:xs.size] do
1770               if (← getFVarLocalDecl xs[i]).binderInfo.isExplicit then
1771                 n := n + 1
1772             return n
1773           let args := args.getElems
1774           if args.size < numExplicitFields then
1775             throwError "invalid constructor {...}, insufficient number of arguments, constructs '{ctor}' has #{numExplicitFields} expl.
1776           let newStx ←
1777             if args.size == numExplicitFields then
1778               `(($ (mkCIdentFrom stx ctor) $(args)*))
1779             else if numExplicitFields == 0 then
1780               throwError "invalid constructor {...}, insufficient number of arguments, constructs '{ctor}' does not have explicit field
1781             else
1782               let extra := args[numExplicitFields-1:args.size]
1783               let newLast ← `(($ [$extra],*))
1784               let newArgs := args[0:numExplicitFields-1].toArray.push newLast
1785               `(($ (mkCIdentFrom stx ctor) $(newArgs)*))

```

```

1786         withMacroExpansion stx newStx $ elabTerm newStx expectedType?
1787         | _ => throwError "invalid constructor {...}, expected type must be an inductive type with only one constructor {indentExpr ex
1788         | none => throwError "invalid constructor {...}, expected type must be known"
1789         | _ => throwUnsupportedSyntax
1790
1791 @[builtinTermElab borrowed] def elabBorrowed : TermElab := fun stx expectedType? =>
1792     match stx with
1793     | `(@& $e) => return markBorrowed (← elabTerm e expectedType?)
1794     | _ => throwUnsupportedSyntax
1795
1796 @[builtinMacro Lean.Parser.Term.show] def expandShow : Macro := fun stx =>
1797     match stx with
1798     | `(show $type from $val)          => let thisId := mkIdentFrom stx `this; `(let_fun $thisId : $type := $val; $thisId)
1799     | `(show $type by $tac:tacticSeq) => `(show $type from by $tac:tacticSeq)
1800     | _                               => Macro.throwUnsupported
1801
1802 @[builtinMacro Lean.Parser.Term.have] def expandHave : Macro := fun stx =>
1803     let mkId (x? : Option Syntax) : Syntax :=
1804         x?.getD <| mkIdentFrom stx `this
1805     match stx with
1806     | `(have $[$x :]? $type from $val $[;]? $body)          => let x := mkId x; `(let_fun $x : $type := $val; $body)
1807     | `(have $[$x :]? $type := $val $[;]? $body)            => let x := mkId x; `(let_fun $x : $type := $val; $body)
1808     | `(have $[$x :]? $type by $tac:tacticSeq $[;]? $body) => `(have $[$x :]? $type from by $tac:tacticSeq; $body)
1809     | _                                                       => Macro.throwUnsupported
1810
1811 @[builtinMacro Lean.Parser.Term.suffices] def expandSuffices : Macro
1812 | `(suffices $[$x :]? $type from $val $[;]? $body)          => `(have $[$x :]? $type from $body; $val)
1813 | `(suffices $[$x :]? $type by $tac:tacticSeq $[;]? $body) => `(have $[$x :]? $type from $body; by $tac:tacticSeq)
1814 | _                                                         => Macro.throwUnsupported
1815
1816 private def elabParserMacroAux (prec : Syntax) (e : Syntax) : TermElabM Syntax := do
1817     let (some declName) ← getDeclName?
1818     | throwError "invalid `leading_parser` macro, it must be used in definitions"
1819     match extractMacroScopes declName with
1820     | { name := Name.str _ s _, scopes := scps, .. } =>
1821         let kind := quote declName
1822         let s     := quote s
1823         let p ← `(Lean.Parser.leadingNode $kind $prec $e)
1824         if scps == [] then
1825             -- TODO simplify the following quotation as soon as we have coercions
1826             `(OrElse.orElse (Lean.Parser.mkAntiquot $s (some $kind)) $p)
1827         else
1828             -- if the parser decl is hidden by hygiene, it doesn't make sense to provide an antiquotation kind
1829             `(OrElse.orElse (Lean.Parser.mkAntiquot $s none) $p)
1830     | _ => throwError "invalid `leading_parser` macro, unexpected declaration name"
1831
1832 @[builtinTermElab «leading_parser»] def elabLeadingParserMacro : TermElab :=

```

```

1833 adaptExpander fun stx => match stx with
1834 | `(leading_parser $e)      => elabParserMacroAux (quote Parser.maxPrec) e
1835 | `(leading_parser : $prec $e) => elabParserMacroAux prec e
1836 | _                          => throwUnsupportedSyntax
1837
1838 private def elabTParserMacroAux (prec lhsPrec : Syntax) (e : Syntax) : TermElabM Syntax := do
1839   let declName? ← getDeclName?
1840   match declName? with
1841   | some declName => let kind := quote declName; `(Lean.Parser.trailingNode $kind $prec $lhsPrec $e)
1842   | none          => throwError "invalid `trailing_parser` macro, it must be used in definitions"
1843
1844 @[builtinTermElab «trailing_parser»] def elabTrailingParserMacro : TermElab :=
1845   adaptExpander fun stx => match stx with
1846   | `(trailing_parser[$:$prec?]?[$:$lhsPrec?]? $e) =>
1847     elabTParserMacroAux (prec?.getD <| quote Parser.maxPrec) (lhsPrec?.getD <| quote 0) e
1848   | _ => throwUnsupportedSyntax
1849
1850 @[builtinTermElab panic] def elabPanic : TermElab := fun stx expectedType? => do
1851   let arg := stx[1]
1852   let pos ← getRefPosition
1853   let env ← getEnv
1854   let stxNew ← match (← getDeclName?) with
1855   | some declName => `(panicWithPosWithDecl $(quote (toString env.mainModule)) $(quote (toString declName)) $(quote pos.line) $(quote pos.column) $arg)
1856   | none => `(panicWithPos $(quote (toString env.mainModule)) $(quote pos.line) $(quote pos.column) $arg)
1857   withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?
1858
1859 @[builtinMacro Lean.Parser.Term.unreachable] def expandUnreachable : Macro := fun stx =>
1860   `(panic! "unreachable code has been reached")
1861
1862 @[builtinMacro Lean.Parser.Term.assert] def expandAssert : Macro := fun stx =>
1863   -- TODO: support for disabling runtime assertions
1864   let cond := stx[1]
1865   let body := stx[3]
1866   match cond.reprint with
1867   | some code => `(if $cond then $body else panic! ("assertion violation: " ++ $(quote code)))
1868   | none => `(if $cond then $body else panic! ("assertion violation"))
1869
1870 @[builtinMacro Lean.Parser.Term.dbgTrace] def expandDbgTrace : Macro := fun stx =>
1871   let arg := stx[1]
1872   let body := stx[3]
1873   if arg.getKind == interpolatedStrKind then
1874     `(dbgTrace (s! $arg) fun _ => $body)
1875   else
1876     `(dbgTrace (toString $arg) fun _ => $body)
1877
1878 @[builtinTermElab «sorry»] def elabSorry : TermElab := fun stx expectedType? => do
1879   logWarning "declaration uses 'sorry'"

```



```

1880 let stxNew ← `(sorryAx _ false)
1881 withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
1882
1883 @[builtinTermElab emptyC] def expandEmptyC : TermElab := fun stx expectedType? => do
1884   let stxNew ← `(EmptyCollection.emptyCollection)
1885   withMacroExpansion stx stxNew $ elabTerm stxNew expectedType?
1886
1887 /-- Return syntax `Prod.mk elems[0] (Prod.mk elems[1] ... (Prod.mk elems[elems.size - 2] elems[elems.size - 1]))` -/
1888 partial def mkPairs (elems : Array Syntax) : MacroM Syntax :=
1889   let rec loop (i : Nat) (acc : Syntax) := do
1890     if i > 0 then
1891       let i := i - 1
1892       let elem := elems[i]
1893       let acc ← `(Prod.mk $elem $acc)
1894       loop i acc
1895     else
1896       pure acc
1897   loop (elems.size - 1) elems.back
1898
1899 private partial def hasCDot : Syntax → Bool
1900 | Syntax.node k args =>
1901   if k == `Lean.Parser.Term.paren then false
1902   else if k == `Lean.Parser.Term.cdor then true
1903   else args.any hasCDot
1904 | _ => false
1905
1906 /--
1907   Auxiliary function for expanding the `` notation.
1908   The extra state `Array Syntax` contains the new binder names.
1909   If `stx` is a ``, we create a fresh identifier, store in the
1910   extra state, and return it. Otherwise, we just return `stx`. -/
1911 private partial def expandCDot : Syntax → StateT (Array Syntax) MacroM Syntax
1912 | stx@(Syntax.node k args) =>
1913   if k == `Lean.Parser.Term.paren then pure stx
1914   else if k == `Lean.Parser.Term.cdor then withFreshMacroScope do
1915     let id ← `(a)
1916     modify fun s => s.push id;
1917     pure id
1918   else do
1919     let args ← args.mapM expandCDot
1920     pure $ Syntax.node k args
1921 | stx => pure stx
1922
1923 /--
1924   Return `some` if succeeded expanding `` notation occurring in
1925   the given syntax. Otherwise, return `none`.
1926   Examples:

```

```

1927 - `· + 1` => `fun _a_1 => _a_1 + 1`
1928 - `f · · b` => `fun _a_1 _a_2 => f _a_1 _a_2 b` -/
1929 def expandCDot? (stx : Syntax) : MacroM (Option Syntax) := do
1930   if hasCDot stx then
1931     let (newStx, binders) ← (expandCDot stx).run #[];
1932     `(fun $binders* => $newStx)
1933   else
1934     pure none
1935
1936 /--
1937   Try to expand `` notation, and if successful elaborate result.
1938   This method is used to elaborate the Lean parentheses notation.
1939   Recall that in Lean the `` notation must be surrounded by parentheses.
1940   We may change this in the future, but right now, here are valid examples
1941   - `(· + 1)`
1942   - `(f (·, 1) ·)`
1943   - `(· + ·)`
1944   - `(f · a b)` -/
1945 private def elabCDot (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
1946   match (← liftMacroM <| expandCDot? stx) with
1947   | some stx' => withMacroExpansion stx stx' (elabTerm stx' expectedType?)
1948   | none      => elabTerm stx expectedType?
1949
1950 /--
1951   Helper method for elaborating terms such as `(.+.)` where a constant name is expected.
1952   This method is usually used to implement tactics that function names as arguments (e.g., `simp`).
1953 -/
1954 def elabCDotFunctionAlias? (stx : Syntax) : TermElabM (Option Expr) := do
1955   let some stx ← liftMacroM <| expandCDotArg? stx | pure none
1956   let stx ← liftMacroM <| expandMacros stx
1957   match stx with
1958   | `(fun $binders* => $f:ident $args*) =>
1959     if binders == args then
1960       try Term.resolveId? f catch _ => return none
1961     else
1962       return none
1963   | _ => return none
1964 where
1965   expandCDotArg? (stx : Syntax) : MacroM (Option Syntax) :=
1966     match stx with
1967     | `(($e)) => Term.expandCDot? e
1968     | _      => Term.expandCDot? stx
1969
1970
1971 @[builtinTermElab paren] def elabParen : TermElab := fun stx expectedType? => do
1972   match stx with
1973   | `(( )) => return Lean.mkConst `Unit.unit

```

```

1974 | `(($e : $type)) =>
1975   let type ← withSynthesize (mayPostpone := true) $ elabType type
1976   let e ← elabCDot e type
1977   ensureHasType type e
1978 | `(($e))      => elabCDot e expectedType?
1979 | `(($e, $es,*)) =>
1980   let pairs ← liftMacroM <| mkPairs ([e] ++ es)
1981   withMacroExpansion stx pairs (elabCDot pairs expectedType?)
1982 | _ => throwError "unexpected parentheses notation"
1983
1984 @[builtinTermElab subst] def elabSubst : TermElab := fun stx expectedType? => do
1985   let expectedType ← tryPostponeIfHasMVars expectedType? "invalid `>` notation"
1986   match stx with
1987   | `($heq > $h) => do
1988     let mut heq ← elabTerm heq none
1989     let heqType ← inferType heq
1990     let heqType ← instantiateMVars heqType
1991     match (← Meta.matchEq? heqType) with
1992     | none => throwError "invalid `>` notation, argument{indentExpr heq}\nhas type{indentExpr heqType}\nequality expected"
1993     | some (α, lhs, rhs) =>
1994       let mut lhs := lhs
1995       let mut rhs := rhs
1996       let mkMotive (typeWithLooseBVar : Expr) :=
1997         withLocalDeclD (← mkFreshUserName `x) α fun x => do
1998           mkLambdaFVars #[x] $ typeWithLooseBVar.instantiate1 x
1999       let mut expectedAbst ← kabstract expectedType rhs
2000       unless expectedAbst.hasLooseBVars do
2001         expectedAbst ← kabstract expectedType lhs
2002       unless expectedAbst.hasLooseBVars do
2003         throwError "invalid `>` notation, expected type{indentExpr expectedType}\ndoes contain equation left-hand-side nor right-hand
2004         heq ← mkEqSymm heq
2005         (lhs, rhs) := (rhs, lhs)
2006       let hExpectedType := expectedAbst.instantiate1 lhs
2007       let h ← withRef h do
2008         let h ← elabTerm h hExpectedType
2009         try
2010           ensureHasType hExpectedType h
2011         catch ex =>
2012           -- if `rhs` occurs in `hType`, we try to apply `heq` to `h` too
2013           let hType ← inferType h
2014           let hTypeAbst ← kabstract hType rhs
2015           unless hTypeAbst.hasLooseBVars do
2016             throw ex
2017           let hTypeNew := hTypeAbst.instantiate1 lhs
2018           unless (← isDefEq hExpectedType hTypeNew) do
2019             throw ex
2020           mkEqNDRRec (← mkMotive hTypeAbst) h (← mkEqSymm heq)

```

```

2021      mkEqNDRec (← mkMotive expectedAbst) h heq
2022 | _ => throwUnsupportedSyntax
2023
2024 @[builtinTermElab stateRefT] def elabStateRefT : TermElab := fun stx _ => do
2025   let σ ← elabType stx[1]
2026   let mut m := stx[2]
2027   if m.getKind == `Lean.Parser.Term.macroDollarArg then
2028     m := m[1]
2029   let m ← elabTerm m (← mkArrow (mkSort levelOne) (mkSort levelOne))
2030   let ω ← mkFreshExprMVar (mkSort levelOne)
2031   let stWorld ← mkAppM `STWorld #[ω, m]
2032   discard <| mkInstMVar stWorld
2033   mkAppM `StateRefT' #[ω, σ, m]
2034
2035 @[builtinTermElab noindex] def elabNoindex : TermElab := fun stx expectedType? => do
2036   let e ← elabTerm stx[1] expectedType?
2037   return DiscrTree.mkNoindexAnnotation e
2038
2039 end Lean.Elab.Term
2040 ::::::::::::::
2041 Elab/Command.lean
2042 ::::::::::::::
2043 /-
2044 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
2045 Released under Apache 2.0 license as described in the file LICENSE.
2046 Authors: Leonardo de Moura
2047 -/
2048 import Lean.Parser.Command
2049 import Lean.ResolveName
2050 import Lean.Meta.Reduce
2051 import Lean.Elab.Log
2052 import Lean.Elab.Term
2053 import Lean.Elab.Binders
2054 import Lean.Elab.SyntheticMVars
2055 import Lean.Elab.DeclModifiers
2056 import Lean.Elab.InfoTree
2057 import Lean.Elab.Open
2058 import Lean.Elab.SetOption
2059
2060 namespace Lean.Elab.Command
2061
2062 structure Scope where
2063   header      : String
2064   opts        : Options := {}
2065   currNamespace : Name := Name.anonymous
2066   openDecls   : List OpenDecl := []
2067   levelNames  : List Name := []

```

```

2068 /-- section variables as `bracketedBinder`s -/
2069 varDecls      : Array Syntax := #[]
2070 /-- Globally unique internal identifiers for the `varDecls` -/
2071 varUIIds      : Array Name  := #[]
2072 deriving Inhabited
2073
2074 structure State where
2075   env          : Environment
2076   messages     : MessageLog := {}
2077   scopes       : List Scope := [{ header := "" }]
2078   nextMacroScope : Nat := firstFrontendMacroScope + 1
2079   maxRecDepth   : Nat
2080   nextInstIdx   : Nat := 1 -- for generating anonymous instance names
2081   ngen          : NameGenerator := {}
2082   infoState     : InfoState := {}
2083   deriving Inhabited
2084
2085 structure Context where
2086   fileName     : String
2087   fileMap      : FileMap
2088   currRecDepth : Nat := 0
2089   cmdPos       : String.Pos := 0
2090   macroStack   : MacroStack := []
2091   currMacroScope : MacroScope := firstFrontendMacroScope
2092   ref          : Syntax := Syntax.missing
2093
2094 abbrev CommandElabCoreM (ε) := ReaderT Context $ StateRefT State $ EIO ε
2095 abbrev CommandElabM := CommandElabCoreM Exception
2096 abbrev CommandElab := Syntax → CommandElabM Unit
2097 abbrev Linter := Syntax → CommandElabM Unit
2098
2099 def mkState (env : Environment) (messages : MessageLog := {}) (opts : Options := {}) : State := {
2100   env      := env
2101   messages := messages
2102   scopes   := [{ header := "", opts := opts }]
2103   maxRecDepth := maxRecDepth.get opts
2104 }
2105
2106 /- Linters should be loadable as plugins, so store in a global IO ref instead of an attribute managed by the
2107    environment (which only contains `import`ed objects). -/
2108 builtin_initialize lintersRef : IO.Ref (Array Linter) ← IO.mkRef #[]
2109
2110 def addLinter (l : Linter) : IO Unit := do
2111   let ls ← lintersRef.get
2112   lintersRef.set (ls.push l)
2113
2114 instance : MonadInfoTree CommandElabM where

```

```

2115   getInfoState      := return (← get).infoState
2116   modifyInfoState f := modify fun s => { s with infoState := f s.infoState }
2117
2118 instance : MonadEnv CommandElabM where
2119   getEnv := do pure (← get).env
2120   modifyEnv f := modify fun s => { s with env := f s.env }
2121
2122 instance : MonadOptions CommandElabM where
2123   getOptions := do pure (← get).scopes.head!.opts
2124
2125 protected def getRef : CommandElabM Syntax :=
2126   return (← read).ref
2127
2128 instance : AddMessageContext CommandElabM where
2129   addMessageContext := addMessageContextPartial
2130
2131 instance : MonadRef CommandElabM where
2132   getRef := Command.getRef
2133   withRef ref x := withReader (fun ctx => { ctx with ref := ref }) x
2134
2135 instance : AddErrorMessageContext CommandElabM where
2136   add ref msg := do
2137     let ctx ← read
2138     let ref := getBetterRef ref ctx.macroStack
2139     let msg ← addMessageContext msg
2140     let msg ← addMacroStack msg ctx.macroStack
2141     return (ref, msg)
2142
2143 def mkMessageAux (ctx : Context) (ref : Syntax) (msgData : MessageData) (severity : MessageSeverity) : Message :=
2144   mkMessageCore ctx.fileName ctx.fileMap msgData severity (ref.getPos?.getD ctx.cmdPos)
2145
2146 private def mkCoreContext (ctx : Context) (s : State) (heartbeats : Nat) : Core.Context :=
2147   let scope      := s.scopes.head!
2148   { options      := scope.opts
2149     currRecDepth := ctx.currRecDepth
2150     maxRecDepth  := s.maxRecDepth
2151     ref          := ctx.ref
2152     currNamespace := scope.currNamespace
2153     openDecls    := scope.openDecls
2154     initHeartbeats := heartbeats }
2155
2156 def liftCoreM {α} (x : CoreM α) : CommandElabM α := do
2157   let s ← get
2158   let ctx ← read
2159   let heartbeats ← IO.getNumHeartbeats (ε := Exception)
2160   let Eα := Except Exception α
2161   let x : CoreM Eα := try let a ← x; pure <| Except.ok a catch ex => pure <| Except.error ex

```

```

2162 let x : EIO Exception (Eα × Core.State) := (ReaderT.run x (mkCoreContext ctx s heartbeats)).run { env := s.env, ngen := s.ngen }
2163 let (ea, coreS) ← liftM x
2164 modify fun s => { s with env := coreS.env, ngen := coreS.ngen }
2165 match ea with
2166 | Except.ok a      => pure a
2167 | Except.error e => throw e
2168
2169 private def ioErrorToMessage (ctx : Context) (ref : Syntax) (err : IO.Error) : Message :=
2170   let ref := getBetterRef ref ctx.macroStack
2171   mkMessageAux ctx ref (toString err) MessageSeverity.error
2172
2173 @[inline] def liftEIO {α} (x : EIO Exception α) : CommandElabM α := liftM x
2174
2175 @[inline] def liftIO {α} (x : IO α) : CommandElabM α := do
2176   let ctx ← read
2177   IO.toEIO (fun (ex : IO.Error) => Exception.error ctx.ref ex.toString) x
2178
2179 instance : MonadLiftT IO CommandElabM where
2180   monadLift := liftIO
2181
2182 def getScope : CommandElabM Scope := do pure (← get).scopes.head!
2183
2184 instance : MonadResolveName CommandElabM where
2185   getCurrNamespace := return (← getScope).currNamespace
2186   getOpenDecls     := return (← getScope).openDecls
2187
2188 instance : MonadLog CommandElabM where
2189   getRef      := getRef
2190   getFileMap  := return (← read).fileMap
2191   getFileName := return (← read).fileName
2192   logMessage msg := do
2193     let currNamespace ← getCurrNamespace
2194     let openDecls ← getOpenDecls
2195     let msg := { msg with data := MessageData.withNamingContext { currNamespace := currNamespace, openDecls := openDecls } msg.data }
2196     modify fun s => { s with messages := s.messages.add msg }
2197
2198 def runLinters (stx : Syntax) : CommandElabM Unit := do
2199   let linters ← lintersRef.get
2200   unless linters.isEmpty do
2201     for linter in linters do
2202       let savedState ← get
2203       try
2204         linter stx
2205       catch ex =>
2206         logException ex
2207       finally
2208         modify fun s => { savedState with messages := s.messages }

```

```

2209
2210 protected def getCurrMacroScope : CommandElabM Nat := do pure (← read).currMacroScope
2211 protected def getMainModule      : CommandElabM Name := do pure (← getEnv).mainModule
2212
2213 @[inline] protected def withFreshMacroScope {α} (x : CommandElabM α) : CommandElabM α := do
2214   let fresh ← modifyGet (fun st => (st.nextMacroScope, { st with nextMacroScope := st.nextMacroScope + 1 }))
2215   withReader (fun ctx => { ctx with currMacroScope := fresh }) x
2216
2217 instance : MonadQuotation CommandElabM where
2218   getCurrMacroScope := Command.getCurrMacroScope
2219   getMainModule      := Command.getMainModule
2220   withFreshMacroScope := Command.withFreshMacroScope
2221
2222 unsafe def mkCommandElabAttributeUnsafe : IO (KeyedDeclsAttribute CommandElab) :=
2223   mkElabAttribute CommandElab `Lean.Elab.Command.commandElabAttribute `builtinCommandElab `commandElab `Lean.Parser.Command `Lean.Elab.C
2224
2225 @[implementedBy mkCommandElabAttributeUnsafe]
2226 constant mkCommandElabAttribute : IO (KeyedDeclsAttribute CommandElab)
2227
2228 builtin_initialize commandElabAttribute : KeyedDeclsAttribute CommandElab ← mkCommandElabAttribute
2229
2230 private def elabCommandUsing (s : State) (stx : Syntax) : List CommandElab → CommandElabM Unit
2231 | [] => throwError "unexpected syntax{indentD stx}"
2232 | (elabFn::elabFns) =>
2233   catchInternalId unsupportedSyntaxExceptionId
2234     (elabFn stx)
2235     (fun _ => do set s; elabCommandUsing s stx elabFns)
2236
2237 /- Elaborate `x` with `stx` on the macro stack -/
2238 @[inline] def withMacroExpansion {α} (beforeStx afterStx : Syntax) (x : CommandElabM α) : CommandElabM α :=
2239   withReader (fun ctx => { ctx with macroStack := { before := beforeStx, after := afterStx } :: ctx.macroStack }) x
2240
2241 instance : MonadMacroAdapter CommandElabM where
2242   getCurrMacroScope := getCurrMacroScope
2243   getNextMacroScope := return (← get).nextMacroScope
2244   setNextMacroScope next := modify fun s => { s with nextMacroScope := next }
2245
2246 instance : MonadRecDepth CommandElabM where
2247   withRecDepth d x := withReader (fun ctx => { ctx with currRecDepth := d }) x
2248   getRecDepth      := return (← read).currRecDepth
2249   getMaxRecDepth   := return (← get).maxRecDepth
2250
2251 register_builtin_option showPartialSyntaxErrors : Bool := {
2252   defValue := false
2253   descr   := "show elaboration errors from partial syntax trees (i.e. after parser recovery)"
2254 }
2255

```



```

2256 @[inline] def withLogging (x : CommandElabM Unit) : CommandElabM Unit := do
2257   try
2258     x
2259   catch ex => match ex with
2260   | Exception.error _ _ => logException ex
2261   | Exception.internal id _ =>
2262     if isAbortExceptionId id then
2263       pure ()
2264     else
2265       let idName ← liftIO <| id.getName;
2266       logError m!"internal exception {idName}"
2267
2268 builtin_initialize registerTraceClass `Elab.command
2269
2270 partial def elabCommand (stx : Syntax) : CommandElabM Unit := do
2271   let mkInfoTree trees := do
2272     let ctx ← read
2273     let s ← get
2274     let scope := s.scopes.head!
2275     let tree := InfoTree.node (Info.ofCommandInfo { stx := stx }) trees
2276     let tree := InfoTree.context {
2277       env := s.env, fileMap := ctx.fileMap, mctx := {}, currNamespace := scope.currNamespace, openDecls := scope.openDecls, options := s
2278     } tree
2279     if checkTraceOption (← getOptions) `Elab.info then
2280       logTrace `Elab.info m!"{← tree.format}"
2281   return tree
2282   let initMsgs ← modifyGet fun st => (st.messages, { st with messages := {} })
2283   withLogging <| withRef stx <| withInfoTreeContext (mkInfoTree := mkInfoTree) <| withIncRecDepth <| withFreshMacroScope do
2284     runLinters stx
2285     match stx with
2286     | Syntax.node k args =>
2287       if k == nullKind then
2288         -- list of commands => elaborate in order
2289         -- The parser will only ever return a single command at a time, but syntax quotations can return multiple ones
2290         args.forM elabCommand
2291       else do
2292         trace `Elab.command fun _ => stx;
2293         let s ← get
2294         let stxNew? ← catchInternalId unsupportedSyntaxExceptionId
2295           (do let newStx ← adaptMacro (getMacros s.env) stx; pure (some newStx))
2296           (fun ex => pure none)
2297         match stxNew? with
2298         | some stxNew => withMacroExpansion stx stxNew <| elabCommand stxNew
2299         | _ =>
2300           let table := (commandElabAttribute.ext.getState s.env).table;
2301           let k := stx.getKind;
2302           match table.find? k with

```

```

2303         | some elabFns => elabCommandUsing s stx elabFns
2304         | none         => throwError "elaboration function for '{k}' has not been implemented"
2305     | _ => throwError "unexpected command"
2306 let mut msgs ← (← get).messages
2307 -- `stx.hasMissing` should imply `initMsgs.hasErrors`, but the latter should be cheaper to check in general
2308 if !showPartialSyntaxErrors.get (← getOptions) && initMsgs.hasErrors && stx.hasMissing then
2309     -- discard elaboration errors, except for a few important and unlikely misleading ones, on parse error
2310     msgs := {msgs.msgs.filter fun msg =>
2311         msg.data.hasTag `Elab.synthPlaceholder || msg.data.hasTag `Tactic.unsolvedGoals}
2312 modify ({ · with messages := initMsgs ++ msgs })
2313
2314 /-- Adapt a syntax transformation to a regular, command-producing elaborator. -/
2315 def adaptExpander (exp : Syntax → CommandElabM Syntax) : CommandElab := fun stx => do
2316     let stx' ← exp stx
2317     withMacroExpansion stx stx' <| elabCommand stx'
2318
2319 private def getVarDecls (s : State) : Array Syntax :=
2320     s.scopes.head!.varDecls
2321
2322 instance {α} : Inhabited (CommandElabM α) where
2323     default := throw arbitrary
2324
2325 private def mkMetaContext : Meta.Context := {
2326     config := { foApprox := true, ctxApprox := true, quasiPatternApprox := true }
2327 }
2328
2329 def getBracketedBinderIds : Syntax → Array Name
2330 | `(bracketedBinder|($ids* $[: $ty]? $(annot?)?)) => ids.map Syntax.getId
2331 | `(bracketedBinder|{$ids* $[: $ty]?})           => ids.map Syntax.getId
2332 | `(bracketedBinder|[$id : $ty])                 => #[id.getId]
2333 | `(bracketedBinder|[$ty])                       => #[]
2334 | _                                               => #[]
2335
2336 private def mkTermContext (ctx : Context) (s : State) (declName? : Option Name) : Term.Context := do
2337     let scope      := s.scopes.head!
2338     let mut sectionVars := {}
2339     for id in scope.varDecls.concatMap getBracketedBinderIds, uid in scope.varUIIds do
2340         sectionVars := sectionVars.insert id uid
2341     { macroStack      := ctx.macroStack
2342       fileName       := ctx.fileName
2343       fileMap        := ctx.fileMap
2344       currMacroScope := ctx.currMacroScope
2345       declName?      := declName?
2346       sectionVars     := sectionVars }
2347
2348 private def mkTermState (scope : Scope) (s : State) : Term.State := {
2349     messages := {}

```

```

2350 levelNames      := scope.levelNames
2351 infoState.enabled := s.infoState.enabled
2352 }
2353
2354 private def addTraceAsMessages (ctx : Context) (log : MessageLog) (traceState : TraceState) : MessageLog :=
2355   traceState.traces.foldl (init := log) fun (log : MessageLog) traceElem =>
2356     let ref := replaceRef traceElem.ref ctx.ref;
2357     let pos := ref.getPos?.getD 0;
2358     log.add (mkMessageCore ctx.fileName ctx.fileMap traceElem.msg MessageSeverity.information pos)
2359
2360 def liftTermElabM {α} (declName? : Option Name) (x : TermElabM α) : CommandElabM α := do
2361   let ctx ← read
2362   let s   ← get
2363   let heartbeats ← IO.getNumHeartbeats (ε := Exception)
2364   -- dbg_trace "heartbeats: {heartbeats}"
2365   let scope := s.scopes.head!
2366   -- We execute `x` with an empty message log. Thus, `x` cannot modify/view messages produced by previous commands.
2367   -- This is useful for implementing `runTermElabM` where we use `Term.resetMessageLog`
2368   let x : MetaM _ := (observing x).run (mkTermContext ctx s declName?) (mkTermState scope s)
2369   let x : CoreM _ := x.run mkMetaContext {}
2370   let x : EIO _   := x.run (mkCoreContext ctx s heartbeats) { env := s.env, ngen := s.ngen, nextMacroScope := s.nextMacroScope }
2371   let ((ea, termS), metaS, coreS) ← liftEIO x
2372   let infoTrees := termS.infoState.trees.map fun tree =>
2373     let tree := tree.substitute termS.infoState.assignment
2374     InfoTree.context {
2375       env := coreS.env, fileMap := ctx.fileMap, mctx := metaS.mctx, currNamespace := scope.currNamespace, openDecls := scope.openDecls,
2376     } tree
2377   modify fun s => { s with
2378     env      := coreS.env
2379     messages := addTraceAsMessages ctx (s.messages ++ termS.messages) coreS.traceState
2380     nextMacroScope := coreS.nextMacroScope
2381     ngen         := coreS.ngen
2382     infoState.trees := s.infoState.trees.append infoTrees
2383   }
2384   match ea with
2385   | Except.ok a      => pure a
2386   | Except.error ex => throw ex
2387
2388 @[inline] def runTermElabM {α} (declName? : Option Name) (elabFn : Array Expr → TermElabM α) : CommandElabM α := do
2389   let scope ← getScope
2390   liftTermElabM declName? <|
2391     Term.withAutoBoundImplicit <|
2392       Term.elabBinders scope.varDecls fun xs => do
2393         -- We need to synthesize postponed terms because this is a checkpoint for the auto-bound implicit feature
2394         -- If we don't use this checkpoint here, then auto-bound implicits in the postponed terms will not be handled correctly.
2395         Term.synthesizeSyntheticMVarsNoPostponing
2396         let mut sectionFVars := {}

```

```

2397     for uid in scope.varUIDs, x in xs do
2398       sectionFVars := sectionFVars.insert uid x
2399     withReader ({ · with sectionFVars := sectionFVars }) do
2400       -- We don't want to store messages produced when elaborating `(getVarDecls s)` because they have already been saved when we el.
2401       -- So, we use `Term.resetMessageLog`.
2402       Term.resetMessageLog
2403       let xs ← Term.addAutoBoundImplicits xs
2404       Term.withoutAutoBoundImplicit <| elabFn xs
2405
2406 @[inline] def catchExceptions (x : CommandElabM Unit) : CommandElabCoreM Empty Unit := fun ctx ref =>
2407   EIO.catchExceptions (withLogging x ctx ref) (fun _ => pure ())
2408
2409 private def liftAttrM {α} (x : AttrM α) : CommandElabM α := do
2410   liftCoreM x
2411
2412 private def addScope (isNewNamespace : Bool) (header : String) (newNamespace : Name) : CommandElabM Unit := do
2413   modify fun s => { s with
2414     env      := s.env.registerNamespace newNamespace,
2415     scopes := { s.scopes.head! with header := header, currNamespace := newNamespace } :: s.scopes
2416   }
2417   pushScope
2418   if isNewNamespace then
2419     activateScoped newNamespace
2420
2421 private def addScopes (isNewNamespace : Bool) : Name → CommandElabM Unit
2422 | Name.anonymous => pure ()
2423 | Name.str p header _ => do
2424   addScopes isNewNamespace p
2425   let currNamespace ← getCurrNamespace
2426   addScope isNewNamespace header (if isNewNamespace then Name.mkStr currNamespace header else currNamespace)
2427 | _ => throwError "invalid scope"
2428
2429 private def addNamespace (header : Name) : CommandElabM Unit :=
2430   addScopes (isNewNamespace := true) header
2431
2432 @[builtinCommandElab «namespace»] def elabNamespace : CommandElab := fun stx =>
2433   match stx with
2434   | `(namespace $n) => addNamespace n.getId
2435   | _               => throwUnsupportedSyntax
2436
2437 @[builtinCommandElab «section»] def elabSection : CommandElab := fun stx =>
2438   match stx with
2439   | `(section $header:ident) => addScopes (isNewNamespace := false) header.getId
2440   | `(section)               => do let currNamespace ← getCurrNamespace; addScope (isNewNamespace := false) "" currNamespace
2441   | _                       => throwUnsupportedSyntax
2442
2443 def getScopes : CommandElabM (List Scope) := do

```

```

2444 pure (← get).scopes
2445
2446 private def checkAnonymousScope : List Scope → Bool
2447 | { header := "", .. } :: _ => true
2448 | _ => false
2449
2450 private def checkEndHeader : Name → List Scope → Bool
2451 | Name.anonymous, _ => true
2452 | Name.str p s _, { header := h, .. } :: scopes => h == s && checkEndHeader p scopes
2453 | _, _ => false
2454
2455 private def popScopes (numScopes : Nat) : CommandElabM Unit :=
2456   for i in [0:numScopes] do
2457     popScope
2458
2459 @[builtinCommandElab «end»] def elabEnd : CommandElab := fun stx => do
2460   let header? := (stx.getArg 1).getOptionalIdent?;
2461   let endSize := match header? with
2462   | none => 1
2463   | some n => n.getNumParts
2464   let scopes ← getScopes
2465   if endSize < scopes.length then
2466     modify fun s => { s with scopes := s.scopes.drop endSize }
2467     popScopes endSize
2468   else -- we keep "root" scope
2469     let n := (← get).scopes.length - 1
2470     modify fun s => { s with scopes := s.scopes.drop n }
2471     popScopes n
2472     throwError "invalid 'end', insufficient scopes"
2473   match header? with
2474   | none =>
2475     unless checkAnonymousScope scopes do
2476       throwError "invalid 'end', name is missing"
2477   | some header =>
2478     unless checkEndHeader header scopes do
2479       addCompletionInfo <| CompletionInfo.endSection stx (scopes.map fun scope => scope.header)
2480       throwError "invalid 'end', name mismatch"
2481
2482 @[inline] def withNamespace {α} (ns : Name) (elabFn : CommandElabM α) : CommandElabM α := do
2483   addNamespace ns
2484   let a ← elabFn
2485   modify fun s => { s with scopes := s.scopes.drop ns.getNumParts }
2486   pure a
2487
2488 @[specialize] def modifyScope (f : Scope → Scope) : CommandElabM Unit :=
2489   modify fun s => { s with
2490     scopes := match s.scopes with

```

```

2491     | h::t => f h :: t
2492     | []  => unreachable!
2493   }
2494
2495 def getLevelNames : CommandElabM (List Name) :=
2496   return (← getScope).levelNames
2497
2498 def addUnivLevel (idStx : Syntax) : CommandElabM Unit := withRef idStx do
2499   let id := idStx.getId
2500   let levelNames ← getLevelNames
2501   if levelNames.elem id then
2502     throwAlreadyDeclaredUniverseLevel id
2503   else
2504     modifyScope fun scope => { scope with levelNames := id :: scope.levelNames }
2505
2506 partial def elabChoiceAux (cmds : Array Syntax) (i : Nat) : CommandElabM Unit :=
2507   if h : i < cmds.size then
2508     let cmd := cmds.get {i, h};
2509     catchInternalId unsupportedSyntaxExceptionId
2510       (elabCommand cmd)
2511     (fun ex => elabChoiceAux cmds (i+1))
2512   else
2513     throwUnsupportedSyntax
2514
2515 @[builtinCommandElab choice] def elbChoice : CommandElab := fun stx =>
2516   elabChoiceAux stx.getArgs 0
2517
2518 @[builtinCommandElab «universe»] def elabUniverse : CommandElab := fun n => do
2519   addUnivLevel n[1]
2520
2521 @[builtinCommandElab «universes»] def elabUniverses : CommandElab := fun n => do
2522   n[1].forArgsM addUnivLevel
2523
2524 @[builtinCommandElab «init_quot»] def elabInitQuot : CommandElab := fun stx => do
2525   match (← getEnv).addDecl Declaration.quotDecl with
2526   | Except.ok env    => setEnv env
2527   | Except.error ex  => throwError (ex.toMessageData (← getOptions))
2528
2529 @[builtinCommandElab «export»] def elabExport : CommandElab := fun stx => do
2530   -- `stx` is of the form (Command.export "export" <namespace> "(" (null <ids>*) ")")
2531   let id := stx[1].getId
2532   let ns ← resolveNamespace id
2533   let currNamespace ← getCurrNamespace
2534   if ns == currNamespace then throwError "invalid 'export', self export"
2535   let env ← getEnv
2536   let ids := stx[3].getArgs
2537   let aliases ← ids.foldlM (init := []) fun (aliases : List (Name × Name)) (idStx : Syntax) => do

```

```

2538   let id := idStx.getId
2539   let declName := ns ++ id
2540   if env.contains declName then
2541     pure <| (currNamespace ++ id, declName) :: aliases
2542   else
2543     withRef idStx <| logUnknownDecl declName
2544     pure aliases
2545   modify fun s => { s with env := aliases.foldl (init := s.env) fun env p => addAlias env p.1 p.2 }
2546
2547 @[builtinCommandElab «open»] def elabOpen : CommandElab := fun n => do
2548   let openDecls ← elabOpenDecl n[1]
2549   modifyScope fun scope => { scope with openDecls := openDecls }
2550
2551 @[builtinCommandElab «variable»] def elabVariable : CommandElab
2552 | `(variable $binders*) => do
2553   -- Try to elaborate `binders` for sanity checking
2554   runTermElabM none fun _ => Term.withAutoBoundImplicit <|
2555     Term.elabBinders binders fun _ => pure ()
2556   let varUIds ← binders.concatMap getBracketedBinderIds |>.mapM (withFreshMacroScope • MonadQuotation.addMacroScope)
2557   modifyScope fun scope => { scope with varDecls := scope.varDecls ++ binders, varUIds := scope.varUIds ++ varUIds }
2558 | _ => throwUnsupportedSyntax
2559
2560 open Meta
2561
2562 @[builtinCommandElab Lean.Parser.Command.check] def elabCheck : CommandElab
2563 | `(#check%$tk $term) => withoutModifyingEnv $ runTermElabM (some `_check) fun _ => do
2564   let e ← Term.elabTerm term none
2565   Term.synthesizeSyntheticMVarsNoPostponing
2566   let (e, _) ← Term.levelMVarToParam (← instantiateMVars e)
2567   let type ← inferType e
2568   unless e.isSyntheticSorry do
2569     logInfoAt tk m! "{e} : {type}"
2570 | _ => throwUnsupportedSyntax
2571
2572 @[builtinCommandElab Lean.Parser.Command.reduce] def elabReduce : CommandElab
2573 | `(#reduce%$tk $term) => withoutModifyingEnv <| runTermElabM (some `_check) fun _ => do
2574   let e ← Term.elabTerm term none
2575   Term.synthesizeSyntheticMVarsNoPostponing
2576   let (e, _) ← Term.levelMVarToParam (← instantiateMVars e)
2577   -- TODO: add options or notation for setting the following parameters
2578   withTheReader Core.Context (fun ctx => { ctx with options := ctx.options.setBool `smartUnfolding false }) do
2579     let e ← withTransparency (mode := TransparencyMode.all) <| reduce e (skipProofs := false) (skipTypes := false)
2580     logInfoAt tk e
2581 | _ => throwUnsupportedSyntax
2582
2583 def hasNoErrorMessages : CommandElabM Bool := do
2584   return !(← get).messages.hasErrors

```

```

2585
2586 def failIfSucceeds (x : CommandElabM Unit) : CommandElabM Unit := do
2587   let resetMessages : CommandElabM MessageLog := do
2588     let s ← get
2589     let messages := s.messages;
2590     modify fun s => { s with messages := {} };
2591     pure messages
2592   let restoreMessages (prevMessages : MessageLog) : CommandElabM Unit := do
2593     modify fun s => { s with messages := prevMessages ++ s.messages.errorsToWarnings }
2594   let prevMessages ← resetMessages
2595   let succeeded ←
2596     try
2597       x
2598       hasNoErrorMessages
2599     catch
2600       | ex@(Exception.error _ _) => do logException ex; pure false
2601       | Exception.internal id _ => do logError (← id.getName); pure false
2602   finally
2603     restoreMessages prevMessages
2604   if succeeded then
2605     throwError "unexpected success"
2606
2607 @[builtinCommandElab «check_failure»] def elabCheckFailure : CommandElab
2608   | `(#check_failure $term) => do
2609     failIfSucceeds <| elabCheck (← `(#check $term))
2610   | _ => throwUnsupportedSyntax
2611
2612 unsafe def elabEvalUnsafe : CommandElab
2613   | `(#eval%$tk $term) => do
2614     let n := `_eval
2615     let ctx ← read
2616     let addAndCompile (value : Expr) : TermElabM Unit := do
2617       let type ← inferType value
2618       let decl := Declaration.defnDecl {
2619         name      := n
2620         levelParams := []
2621         type      := type
2622         value     := value
2623         hints     := ReducibilityHints.opaque
2624         safety    := DefinitionSafety.unsafe
2625       }
2626       Term.ensureNoUnassignedMVars decl
2627       addAndCompile decl
2628     let elabMetaEval : CommandElabM Unit := runTermElabM (some n) fun _ => do
2629       let e ← Term.elabTerm term none
2630       Term.synthesizeSyntheticMVarsNoPostponing
2631       let e ← withLocalDeclD `env (mkConst ``Lean.Environment) fun env =>

```



```

2632         withLocalDeclD `opts (mkConst ``Lean.Options) fun opts => do
2633             let e ← mkAppM ``Lean.runMetaEval #[env, opts, e];
2634             mkLambdaFVars #[env, opts] e
2635         let env ← getEnv
2636         let opts ← getOptions
2637         let act ← try addAndCompile e; evalConst (Environment → Options → IO (String × Except IO.Error Environment)) n finally setEnv env
2638         let (out, res) ← act env opts -- we execute `act` using the environment
2639         logInfoAt tk out
2640         match res with
2641         | Except.error e => throwError e.toString
2642         | Except.ok env => do setEnv env; pure ()
2643     let elabEval : CommandElabM Unit := runTermElabM (some n) fun _ => do
2644         -- fall back to non-meta eval if MetaEval hasn't been defined yet
2645         -- modify e to `runEval e`
2646         let e ← Term.elabTerm term none
2647         let e := mkSimpleThunk e
2648         Term.synthesizeSyntheticMVarsNoPostponing
2649         let e ← mkAppM ``Lean.runEval #[e]
2650         let env ← getEnv
2651         let act ← try addAndCompile e; evalConst (IO (String × Except IO.Error Unit)) n finally setEnv env
2652         let (out, res) ← liftM (m := IO) act
2653         logInfoAt tk out
2654         match res with
2655         | Except.error e => throwError e.toString
2656         | Except.ok _ => pure ()
2657     if (← getEnv).contains ``Lean.MetaEval then do
2658         elabMetaEval
2659     else
2660         elabEval
2661 | _ => throwUnsupportedSyntax
2662
2663 @[builtinCommandElab «eval», implementedBy elabEvalUnsafe]
2664 constant elabEval : CommandElab
2665
2666 @[builtinCommandElab «synth»] def elabSynth : CommandElab := fun stx => do
2667     let term := stx[1]
2668     withoutModifyingEnv <| runTermElabM `_synth_cmd fun _ => do
2669         let inst ← Term.elabTerm term none
2670         Term.synthesizeSyntheticMVarsNoPostponing
2671         let inst ← instantiateMVars inst
2672         let val ← synthInstance inst
2673         logInfo val
2674         pure ()
2675
2676 @[builtinCommandElab «set_option»] def elabSetOption : CommandElab := fun stx => do
2677     let options ← Elab.elabSetOption stx[1] stx[2]
2678     modify fun s => { s with maxRecDepth := maxRecDepth.get options }

```

```

2679   modifyScope fun scope => { scope with opts := options }
2680
2681 @[builtinMacro Lean.Parser.Command.«in»] def expandInCmd : Macro := fun stx => do
2682   let cmd₁ := stx[0]
2683   let cmd₂ := stx[2]
2684   `(section $cmd₁:command $cmd₂:command end)
2685
2686 def expandDeclId (declId : Syntax) (modifiers : Modifiers) : CommandElabM ExpandDeclIdResult := do
2687   let currNamespace ← getCurrNamespace
2688   let currLevelNames ← getLevelNames
2689   Lean.Elab.expandDeclId currNamespace currLevelNames declId modifiers
2690
2691 end Elab.Command
2692
2693 export Elab.Command (Linter addLinter)
2694
2695 end Lean
2696 ::::::::::::::
2697 Elab/Declaration.lean
2698 ::::::::::::::
2699 /-
2700 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
2701 Released under Apache 2.0 license as described in the file LICENSE.
2702 Authors: Leonardo de Moura, Sebastian Ullrich
2703 -/
2704 import Lean.Util.CollectLevelParams
2705 import Lean.Elab.DeclUtil
2706 import Lean.Elab.DefView
2707 import Lean.Elab.Inductive
2708 import Lean.Elab.Structure
2709 import Lean.Elab.MutualDef
2710 import Lean.Elab.DeclarationRange
2711 namespace Lean.Elab.Command
2712
2713 open Meta
2714
2715 /- Auxiliary function for `expandDeclNamespace?` -/
2716 def expandDeclIdNamespace? (declId : Syntax) : Option (Name × Syntax) :=
2717   let (id, optUnivDeclStx) := expandDeclIdCore declId
2718   let scpView := extractMacroScopes id
2719   match scpView.name with
2720   | Name.str Name.anonymous s _ => none
2721   | Name.str pre s _           =>
2722     let nameNew := { scpView with name := Name.mkSimple s }.review
2723     if declId.isIdent then
2724       some (pre, mkIdentFrom declId nameNew)
2725     else

```

```

2726     some (pre, declId.setArg 0 (mkIdentFrom declId nameNew))
2727 | _ => none
2728
2729 /- given declarations such as `@[...] def Foo.Bla.f ...` return `some (Foo.Bla, @[...] def f ...)` -/
2730 def expandDeclNamespace? (stx : Syntax) : Option (Name × Syntax) :=
2731   if !stx.isOfKind `Lean.Parser.Command.declaration then none
2732   else
2733     let decl := stx[1]
2734     let k := decl.getKind
2735     if k == `Lean.Parser.Command.abbrev ||
2736        k == `Lean.Parser.Command.def ||
2737        k == `Lean.Parser.Command.theorem ||
2738        k == `Lean.Parser.Command.constant ||
2739        k == `Lean.Parser.Command.axiom ||
2740        k == `Lean.Parser.Command.inductive ||
2741        k == `Lean.Parser.Command.classInductive ||
2742        k == `Lean.Parser.Command.structure then
2743       match expandDeclIdNamespace? decl[1] with
2744       | some (ns, declId) => some (ns, stx.setArg 1 (decl.setArg 1 declId))
2745       | none => none
2746     else if k == `Lean.Parser.Command.instance then
2747       let optDeclId := decl[3]
2748       if optDeclId.isNone then none
2749       else match expandDeclIdNamespace? optDeclId[0] with
2750       | some (ns, declId) => some (ns, stx.setArg 1 (decl.setArg 3 (optDeclId.setArg 0 declId)))
2751       | none => none
2752     else
2753       none
2754
2755 def elabAxiom (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
2756   -- leading_parser "axiom" >> declId >> declSig
2757   let declId := stx[1]
2758   let (binders, typeStx) := expandDeclSig stx[2]
2759   let scopeLevelNames ← getLevelNames
2760   let (name, declName, allUserLevelNames) ← expandDeclId declId modifiers
2761   addDeclarationRanges declName stx
2762   runTermElabM declName fun vars => Term.withLevelNames allUserLevelNames $ Term.elabBinders binders.getArgs fun xs => do
2763     Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.beforeElaboration
2764     let type ← Term.elabType typeStx
2765     Term.synthesizeSyntheticMVarsNoPostponing
2766     let type ← instantiateMVars type
2767     let type ← mkForallFVars xs type
2768     let type ← mkForallFVars vars type (usedOnly := true)
2769     let (type, _) ← Term.levelMVarToParam type
2770     let usedParams := collectLevelParams {} type |>.params
2771     match sortDeclLevelParams scopeLevelNames allUserLevelNames usedParams with
2772     | Except.error msg => throwErrorAt stx msg

```

```

2773 | Except.ok levelParams =>
2774   let decl := Declaration.axiomDecl {
2775     name      := declName,
2776     levelParams := levelParams,
2777     type      := type,
2778     isUnsafe   := modifiers.isUnsafe
2779   }
2780   Term.ensureNoUnassignedMVars decl
2781   addDecl decl
2782   Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.afterTypeChecking
2783   if isExtern (← getEnv) declName then
2784     compileDecl decl
2785   Term.applyAttributesAt declName modifiers.attrs AttributeApplicationTime.afterCompilation
2786
2787 /-
2788 leading_parser "inductive " >> declId >> optDeclSig >> optional "!=" >> many ctor
2789 leading_parser atomic (group ("class " >> "inductive ")) >> declId >> optDeclSig >> optional "!=" >> many ctor >> optDeriving
2790 -/
2791 private def inductiveSyntaxToView (modifiers : Modifiers) (decl : Syntax) : CommandElabM InductiveView := do
2792   checkValidInductiveModifier modifiers
2793   let (binders, type?) := expandOptDeclSig decl[2]
2794   let declId           := decl[1]
2795   let {name, declName, levelNames} ← expandDeclId declId modifiers
2796   addDeclarationRanges declName decl
2797   let ctors          ← decl[4].getArgs.mapM fun ctor => withRef ctor do
2798     -- def ctor := leading_parser " | " >> declModifiers >> ident >> optional inferMod >> optDeclSig
2799     let ctorModifiers ← elabModifiers ctor[1]
2800     if ctorModifiers.isPrivate && modifiers.isPrivate then
2801       throwError "invalid 'private' constructor in a 'private' inductive datatype"
2802     if ctorModifiers.isProtected && modifiers.isPrivate then
2803       throwError "invalid 'protected' constructor in a 'private' inductive datatype"
2804     checkValidCtorModifier ctorModifiers
2805     let ctorName := ctor.getIdAt 2
2806     let ctorName := declName ++ ctorName
2807     let ctorName ← withRef ctor[2] $ applyVisibility ctorModifiers.visibility ctorName
2808     let inferMod := !ctor[3].isNone
2809     let (binders, type?) := expandOptDeclSig ctor[4]
2810     addDocString' ctorName ctorModifiers.docString?
2811     addAuxDeclarationRanges ctorName ctor ctor[2]
2812     pure { ref := ctor, modifiers := ctorModifiers, declName := ctorName, inferMod := inferMod, binders := binders, type? := type? : Cto
2813   let classes ← getOptDerivingClasses decl[5]
2814   pure {
2815     ref           := decl
2816     modifiers     := modifiers
2817     shortDeclName := name
2818     declName      := declName
2819     levelNames    := levelNames

```

```

2820     binders          := binders
2821     type?             := type?
2822     ctors             := ctors
2823     derivingClasses := classes
2824 }
2825
2826 private def classInductiveSyntaxToView (modifiers : Modifiers) (decl : Syntax) : CommandElabM InductiveView :=
2827   inductiveSyntaxToView modifiers decl
2828
2829 def elabInductive (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
2830   let v ← inductiveSyntaxToView modifiers stx
2831   elabInductiveViews #[v]
2832
2833 def elabClassInductive (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
2834   let modifiers := modifiers.addAttribute { name := `class }
2835   let v ← classInductiveSyntaxToView modifiers stx
2836   elabInductiveViews #[v]
2837
2838 @[builtinCommandElab declaration]
2839 def elabDeclaration : CommandElab := fun stx =>
2840   match expandDeclNamespace? stx with
2841   | some (ns, newStx) => do
2842     let ns := mkIdentFrom stx ns
2843     let newStx ← `(namespace $ns:ident $newStx end $ns:ident)
2844     withMacroExpansion stx newStx $ elabCommand newStx
2845   | none => do
2846     let modifiers ← elabModifiers stx[0]
2847     let decl      := stx[1]
2848     let declKind := decl.getKind
2849     if declKind == `Lean.Parser.Command.«axiom» then
2850       elabAxiom modifiers decl
2851     else if declKind == `Lean.Parser.Command.«inductive» then
2852       elabInductive modifiers decl
2853     else if declKind == `Lean.Parser.Command.classInductive then
2854       elabClassInductive modifiers decl
2855     else if declKind == `Lean.Parser.Command.«structure» then
2856       elabStructure modifiers decl
2857     else if isDefLike decl then
2858       elabMutualDef #[stx]
2859     else
2860       throwError "unexpected declaration"
2861
2862 /- Return true if all elements of the mutual-block are inductive declarations. -/
2863 private def isMutualInductive (stx : Syntax) : Bool :=
2864   stx[1].getArgs.all fun elem =>
2865     let decl      := elem[1]
2866     let declKind := decl.getKind

```

```

2867     declKind == `Lean.Parser.Command.inductive
2868
2869 private def elabMutualInductive (elems : Array Syntax) : CommandElabM Unit := do
2870   let views ← elems.mapM fun stx => do
2871     let modifiers ← elabModifiers stx[0]
2872     inductiveSyntaxToView modifiers stx[1]
2873   elabInductiveViews views
2874
2875 /- Return true if all elements of the mutual-block are definitions/theorems/abbrevs. -/
2876 private def isMutualDef (stx : Syntax) : Bool :=
2877   stx[1].getArgs.all fun elem =>
2878     let decl := elem[1]
2879     isDefLike decl
2880
2881 private def isMutualPreambleCommand (stx : Syntax) : Bool :=
2882   let k := stx.getKind
2883   k == `Lean.Parser.Command.variable ||
2884   k == `Lean.Parser.Command.variables ||
2885   k == `Lean.Parser.Command.universe ||
2886   k == `Lean.Parser.Command.universes ||
2887   k == `Lean.Parser.Command.check ||
2888   k == `Lean.Parser.Command.set_option ||
2889   k == `Lean.Parser.Command.open
2890
2891 private partial def splitMutualPreamble (elems : Array Syntax) : Option (Array Syntax × Array Syntax) :=
2892   let rec loop (i : Nat) : Option (Array Syntax × Array Syntax) :=
2893     if h : i < elems.size then
2894       let elem := elems.get (i, h)
2895       if isMutualPreambleCommand elem then
2896         loop (i+1)
2897       else if i == 0 then
2898         none -- `mutual` block does not contain any preamble commands
2899       else
2900         some (elems[0:i], elems[i:elems.size])
2901     else
2902       none -- a `mutual` block containing only preamble commands is not a valid `mutual` block
2903   loop 0
2904
2905 @[builtinMacro Lean.Parser.Command.mutual]
2906 def expandMutualNamespace : Macro := fun stx => do
2907   let mut ns? := none
2908   let mut elemsNew := #[]
2909   for elem in stx[1].getArgs do
2910     match ns?, expandDeclNamespace? elem with
2911     | _, none => elemsNew := elemsNew.push elem
2912     | none, some (ns, elem) => ns? := some ns; elemsNew := elemsNew.push elem
2913     | some nsCurr, some (nsNew, elem) =>

```

```

2914     if nsCurr == nsNew then
2915       elemsNew := elemsNew.push elem
2916     else
2917       Macro.throwErrorAt elem s!"conflicting namespaces in mutual declaration, using namespace '{nsNew}', but used '{nsCurr}' in previ
2918 match ns? with
2919 | some ns =>
2920   let ns := mkIdentFrom stx ns
2921   let stxNew := stx.setArg 1 (mkNullNode elemsNew)
2922   `(namespace $ns:ident $stxNew end $ns:ident)
2923 | none => Macro.throwUnsupported
2924
2925 @[builtinMacro Lean.Parser.Command.mutual]
2926 def expandMutualElement : Macro := fun stx => do
2927   let mut elemsNew := #[]
2928   let mut modified := false
2929   for elem in stx[1].getArgs do
2930     match (← expandMacro? elem) with
2931     | some elemNew => elemsNew := elemsNew.push elemNew; modified := true
2932     | none => elemsNew := elemsNew.push elem
2933   if modified then
2934     pure $ stx.setArg 1 (mkNullNode elemsNew)
2935   else
2936     Macro.throwUnsupported
2937
2938 @[builtinMacro Lean.Parser.Command.mutual]
2939 def expandMutualPreamble : Macro := fun stx =>
2940   match splitMutualPreamble stx[1].getArgs with
2941   | none => Macro.throwUnsupported
2942   | some (preamble, rest) => do
2943     let secCmd ← `(section)
2944     let newMutual := stx.setArg 1 (mkNullNode rest)
2945     let endCmd ← `(end)
2946     pure $ mkNullNode ([secCmd] ++ preamble ++ [newMutual] ++ [endCmd])
2947
2948 @[builtinCommandElab «mutual»]
2949 def elabMutual : CommandElab := fun stx => do
2950   if isMutualInductive stx then
2951     elabMutualInductive stx[1].getArgs
2952   else if isMutualDef stx then
2953     elabMutualDef stx[1].getArgs
2954   else
2955     throwError "invalid mutual block"
2956
2957 /- leading_parser "attribute " >> "[" >> sepBy1 (eraseAttr <|> Term.attrInstance) ", " >> "]" >> many1 ident -/
2958 @[builtinCommandElab «attribute»] def elabAttr : CommandElab := fun stx => do
2959   let mut attrInsts := #[]
2960   let mut toErase := #[]

```

```

2961 for attrKindStx in stx[2].getSepArgs do
2962   if attrKindStx.getKind == ``Lean.Parser.Command.eraseAttr then
2963     let attrName := attrKindStx[1].getId.eraseMacroScopes
2964     unless isAttribute (← getEnv) attrName do
2965       throwError "unknown attribute [{attrName}]"
2966     toErase := toErase.push attrName
2967   else
2968     attrInsts := attrInsts.push attrKindStx
2969 let attrs ← elabAttrs attrInsts
2970 let idents := stx[4].getArgs
2971 for ident in idents do withRef ident <| liftTermElabM none do
2972   let declName ← resolveGlobalConstNoOverloadWithInfo ident
2973   Term.applyAttributes declName attrs
2974   for attrName in toErase do
2975     Attribute.erase declName attrName
2976
2977 def expandInitCmd (builtin : Bool) : Macro := fun stx =>
2978   let optHeader := stx[1]
2979   let doSeq      := stx[2]
2980   let attrId     := mkIdentFrom stx $ if builtin then `builtinInit else `init
2981   if optHeader.isNone then
2982     `([${attrId:ident}]def initFn : IO Unit := do $doSeq)
2983   else
2984     let id      := optHeader[0]
2985     let type    := optHeader[1][1]
2986     `(def initFn : IO $type := do $doSeq
2987       @[${attrId:ident} initFn]constant $id : $type)
2988
2989 @[builtinMacro Lean.Parser.Command.«initialize»] def expandInitialize : Macro :=
2990   expandInitCmd (builtin := false)
2991
2992 @[builtinMacro Lean.Parser.Command.«builtin_initialize»] def expandBuiltinInitialize : Macro :=
2993   expandInitCmd (builtin := true)
2994
2995 end Lean.Elab.Command
2996 ::::::::::::::
2997 Elab/DeclarationRange.lean
2998 ::::::::::::::
2999 /-
3000 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
3001 Released under Apache 2.0 license as described in the file LICENSE.
3002 Authors: Leonardo de Moura
3003 -/
3004 import Lean.DeclarationRange
3005 import Lean.Elab.Log
3006 import Lean.Data.Lsp.Utf16
3007

```



```

3008 namespace Lean.Elab
3009
3010 def getDeclarationRange [Monad m] [MonadFileMap m] (stx : Syntax) : m DeclarationRange := do
3011   let fileMap ← getFileMap
3012   let pos      := stx.getPos?.getD 0
3013   let endPos   := stx.getTailPos?.getD pos |> fileMap.toPosition
3014   let pos      := pos |> fileMap.toPosition
3015   return {
3016     pos      := pos
3017     charUtf16 := fileMap.leanPosToLspPos pos |>.character
3018     endPos    := endPos
3019     endCharUtf16 := fileMap.leanPosToLspPos endPos |>.character
3020   }
3021
3022 /--
3023   For most builtin declarations, the selection range is just its name, which is stored in the second position.
3024   Example:
3025   ```
3026   "def " >> declId >> optDeclSig >> declVal
3027   ```
3028   For instances, we use the whole header since the name is optional.
3029   This function converts the given `Syntax` into one that represents its "selection range".
3030 -/
3031 def getDeclarationSelectionRef (stx : Syntax) : Syntax :=
3032   if stx.getKind == ``Parser.Command.«instance» then
3033     stx.setArg 5 mkNullNode
3034   else
3035     stx[1]
3036
3037 /--
3038   Store the `range` and `selectionRange` for `declName` where `stx` is the whole syntax object describing `declName`.
3039   This method is for the builtin declarations only.
3040   User-defined commands should use `Lean.addDeclarationRanges` to store this information for their commands. -/
3041 def addDeclarationRanges [Monad m] [MonadEnv m] [MonadFileMap m] (declName : Name) (stx : Syntax) : m Unit := do
3042   if stx.getKind == ``Parser.Command.«example» then
3043     return ()
3044   else
3045     Lean.addDeclarationRanges declName {
3046       range      := (← getDeclarationRange stx)
3047       selectionRange := (← getDeclarationRange (getDeclarationSelectionRef stx))
3048     }
3049
3050 /-- Auxiliary method for recording ranges for auxiliary declarations (e.g., fields, nested declarations, etc. -/
3051 def addAuxDeclarationRanges [Monad m] [MonadEnv m] [MonadFileMap m] (declName : Name) (stx : Syntax) (header : Syntax) : m Unit := do
3052   Lean.addDeclarationRanges declName {
3053     range      := (← getDeclarationRange stx)
3054     selectionRange := (← getDeclarationRange header)

```

```

3055 }
3056
3057 end Lean.Elab
3058 ::::::::::::::
3059 Elab/DeclModifiers.lean
3060 ::::::::::::::
3061 /-
3062 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
3063 Released under Apache 2.0 license as described in the file LICENSE.
3064 Authors: Leonardo de Moura, Sebastian Ullrich
3065 -/
3066 import Lean.Modifiers
3067 import Lean.DocString
3068 import Lean.Elab.Attributes
3069 import Lean.Elab.Exception
3070 import Lean.Elab.DeclUtil
3071
3072 namespace Lean.Elab
3073
3074 def checkNotAlreadyDeclared {m} [Monad m] [MonadEnv m] [MonadError m] (declName : Name) : m Unit := do
3075   let env ← getEnv
3076   if env.contains declName then
3077     match privateToUserName? declName with
3078     | none      => throwError "'{declName}' has already been declared"
3079     | some declName => throwError "private declaration '{declName}' has already been declared"
3080   if env.contains (mkPrivateName env declName) then
3081     throwError "a private declaration '{declName}' has already been declared"
3082   match privateToUserName? declName with
3083   | none => pure ()
3084   | some declName =>
3085     if env.contains declName then
3086       throwError "a non-private declaration '{declName}' has already been declared"
3087
3088 inductive Visibility where
3089   | regular | «protected» | «private»
3090   deriving Inhabited
3091
3092 instance : ToString Visibility := {fun
3093   | Visibility.regular      => "regular"
3094   | Visibility.«private»   => "private"
3095   | Visibility.«protected» => "protected"}
3096
3097 structure Modifiers where
3098   docString?      : Option String := none
3099   visibility       : Visibility := Visibility.regular
3100   isNoncomputable : Bool := false
3101   isPartial       : Bool := false

```

```

3102 isUnsafe      : Bool := false
3103 attrs         : Array Attribute := #[]
3104 deriving Inhabited
3105
3106 def Modifiers.isPrivate : Modifiers → Bool
3107 | { visibility := Visibility.private, .. } => true
3108 | _                                     => false
3109
3110 def Modifiers.isProtected : Modifiers → Bool
3111 | { visibility := Visibility.protected, .. } => true
3112 | _                                     => false
3113
3114 def Modifiers.addAttribute (modifiers : Modifiers) (attr : Attribute) : Modifiers :=
3115   { modifiers with attrs := modifiers.attrs.push attr }
3116
3117 instance : ToFormat Modifiers := (fun m =>
3118   let components : List Format :=
3119     (match m.docString? with
3120     | some str => [f!"/--{str}-/"]
3121     | none     => [])
3122   ++ (match m.visibility with
3123   | Visibility.regular    => []
3124   | Visibility.protected => [f!"protected"]
3125   | Visibility.private   => [f!"private"])
3126   ++ (if m.isNoncomputable then [f!"noncomputable"] else [])
3127   ++ (if m.isPartial then [f!"partial"] else [])
3128   ++ (if m.isUnsafe then [f!"unsafe"] else [])
3129   ++ m.attrs.toList.map (fun attr => fmt attr)
3130   Format.bracket "{" (Format.joinSep components ("," ++ Format.line)) "}")
3131
3132 instance : ToString Modifiers := (toString ◦ format)
3133
3134 def expandOptDocComment? [Monad m] [MonadError m] (optDocComment : Syntax) : m (Option String) :=
3135   match optDocComment.getOptional? with
3136   | none     => pure none
3137   | some s => match s[1] with
3138   | Syntax.atom _ val => pure (some (val.extract 0 (val.bsize - 2)))
3139   | _                 => throwErrorAt s "unexpected doc string{indentD s[1]}"
3140
3141 section Methods
3142
3143 variable [Monad m] [MonadEnv m] [MonadResolveName m] [MonadError m] [MonadMacroAdapter m] [MonadRecDepth m]
3144
3145 def elabModifiers (stx : Syntax) : m Modifiers := do
3146   let docCommentStx := stx[0]
3147   let attrsStx      := stx[1]
3148   let visibilityStx := stx[2]

```

```

3149 let noncompStx      := stx[3]
3150 let unsafeStx        := stx[4]
3151 let partialStx       := stx[5]
3152 let docString? ← match docCommentStx.getOptional? with
3153   | none   => pure none
3154   | some s => match s[1] with
3155     | Syntax.atom _ val => pure (some (val.extract 0 (val.bsize - 2)))
3156     | _                => throwErrorAt s "unexpected doc string{indentD s[1]}"
3157 let visibility ← match visibilityStx.getOptional? with
3158   | none   => pure Visibility.regular
3159   | some v =>
3160     let kind := v.getKind
3161     if kind == `Lean.Parser.Command.private then pure Visibility.private
3162     else if kind == `Lean.Parser.Command.protected then pure Visibility.protected
3163     else throwErrorAt v "unexpected visibility modifier"
3164 let attrs ← match attrsStx.getOptional? with
3165   | none      => pure #[]
3166   | some attrs => elabDeclAttrs attrs
3167 pure {
3168   docString?      := docString?,
3169   visibility       := visibility,
3170   isPartial       := !partialStx.isNone,
3171   isUnsafe        := !unsafeStx.isNone,
3172   isNoncomputable := !noncompStx.isNone,
3173   attrs           := attrs
3174 }
3175
3176 def applyVisibility (visibility : Visibility) (declName : Name) : m Name := do
3177   match visibility with
3178   | Visibility.private =>
3179     let env ← getEnv
3180     let declName := mkPrivateName env declName
3181     checkNotAlreadyDeclared declName
3182     pure declName
3183   | Visibility.protected =>
3184     checkNotAlreadyDeclared declName
3185     let env ← getEnv
3186     let env := addProtected env declName
3187     setEnv env
3188     pure declName
3189   | _ =>
3190     checkNotAlreadyDeclared declName
3191     pure declName
3192
3193 def mkDeclName (currNamespace : Name) (modifiers : Modifiers) (shortName : Name) : m (Name × Name) := do
3194   let name := (extractMacroScopes shortName).name
3195   unless name.isAtomic || isFreshInstanceName name do

```

```

3196     throwError "atomic identifier expected '{shortName}'"
3197   let declName := currNamespace ++ shortName
3198   let declName ← applyVisibility modifiers.visibility declName
3199   match modifiers.visibility with
3200   | Visibility.protected =>
3201     match currNamespace with
3202     | Name.str _ s _ => pure (declName, Name.mkSimple s ++ shortName)
3203     | _ => throwError "protected declarations must be in a namespace"
3204   | _ => pure (declName, shortName)
3205
3206 /-
3207   `declId` is of the form
3208   ```
3209   leading_parser ident >> optional ("{" >> sepBy1 ident ", " >> "}")
3210   ```
3211   but we also accept a single identifier to users to make macro writing more convenient .
3212 -/
3213 def expandDeclIdCore (declId : Syntax) : Name × Syntax :=
3214   if declId.isIdent then
3215     (declId.getId, mkNullNode)
3216   else
3217     let id          := declId[0].getId
3218     let optUnivDeclStx := declId[1]
3219     (id, optUnivDeclStx)
3220
3221 structure ExpandDeclIdResult where
3222   shortName : Name
3223   declName  : Name
3224   levelNames : List Name
3225
3226 def expandDeclId (currNamespace : Name) (currLevelNames : List Name) (declId : Syntax) (modifiers : Modifiers) : m ExpandDeclIdResult :=
3227   -- ident >> optional ("{" >> sepBy1 ident ", " >> "}")
3228   let (shortName, optUnivDeclStx) := expandDeclIdCore declId
3229   let levelNames ←
3230     if optUnivDeclStx.isNone then
3231       pure currLevelNames
3232     else
3233       let extraLevels := optUnivDeclStx[1].getArgs.getEvenElems
3234       extraLevels.foldlM
3235         (fun levelNames idStx =>
3236           let id := idStx.getId
3237           if levelNames.elem id then
3238             withRef idStx $ throwAlreadyDeclaredUniverseLevel id
3239           else
3240             pure (id :: levelNames))
3241         currLevelNames
3242   let (declName, shortName) ← withRef declId $ mkDeclName currNamespace modifiers shortName

```

```

3243   addDocString' declName modifiers.docString?
3244   pure { shortName := shortName, declName := declName, levelNames := levelNames }
3245
3246 end Methods
3247
3248 end Lean.Elab
3249 ::::::::::::::
3250 Elab/DeclUtil.lean
3251 ::::::::::::::
3252 /-
3253 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
3254 Released under Apache 2.0 license as described in the file LICENSE.
3255 Authors: Leonardo de Moura, Sebastian Ullrich
3256 -/
3257 import Lean.Meta.ExprDefEq
3258
3259 namespace Lean.Meta
3260
3261 def forallTelescopeCompatibleAux {α} (k : Array Expr → Expr → Expr → MetaM α) : Nat → Expr → Expr → Array Expr → MetaM α
3262 | 0, type₁, type₂, xs => k xs type₁ type₂
3263 | i+1, type₁, type₂, xs => do
3264   let type₁ ← whnf type₁
3265   let type₂ ← whnf type₂
3266   match type₁, type₂ with
3267   | Expr.forallE n₁ d₁ b₁ c₁, Expr.forallE n₂ d₂ b₂ c₂ =>
3268     unless n₁ == n₂ do
3269       throwError "parameter name mismatch '{n₁}', expected '{n₂}'"
3270     unless (← isDefEq d₁ d₂) do
3271       throwError "parameter '{n₁}' {← mkHasTypeButIsExpectedMsg d₁ d₂}"
3272     unless c₁.binderInfo == c₂.binderInfo do
3273       throwError "binder annotation mismatch at parameter '{n₁}'"
3274     withLocalDecl n₁ c₁.binderInfo d₁ fun x =>
3275       let type₁ := b₁.instantiate1 x
3276       let type₂ := b₂.instantiate1 x
3277       forallTelescopeCompatibleAux k i type₁ type₂ (xs.push x)
3278   | _, _ => throwError "unexpected number of parameters"
3279
3280 /- Given two forall-expressions `type₁` and `type₂`, ensure the first `numParams` parameters are compatible, and
3281    then execute `k` with the parameters and remaining types. -/
3282 @[inline] def forallTelescopeCompatible {α m} [Monad m] [MonadControlT MetaM m] (type₁ type₂ : Expr) (numParams : Nat) (k : Array Expr →
3283   controlAt MetaM fun runInBase =>
3284     forallTelescopeCompatibleAux (fun xs type₁ type₂ => runInBase $ k xs type₁ type₂) numParams type₁ type₂ #[]
3285
3286 end Meta
3287
3288 namespace Elab
3289

```

```

3290 def expandOptDeclSig (stx : Syntax) : Syntax × Option Syntax :=
3291   -- many Term.bracketedBinder >> Term.optType
3292   let binders := stx[0]
3293   let optType := stx[1] -- optional (leading_parser " : " >> termParser)
3294   if optType.isNone then
3295     (binders, none)
3296   else
3297     let typeSpec := optType[0]
3298     (binders, some typeSpec[1])
3299
3300 def expandDeclSig (stx : Syntax) : Syntax × Syntax :=
3301   -- many Term.bracketedBinder >> Term.typeSpec
3302   let binders := stx[0]
3303   let typeSpec := stx[1]
3304   (binders, typeSpec[1])
3305
3306 def mkFreshInstanceName (env : Environment) (nextIdx : Nat) : Name :=
3307   (env.mainModule ++ `_instance).appendIndexAfter nextIdx
3308
3309 def isFreshInstanceName (name : Name) : Bool :=
3310   match name with
3311   | Name.str _ s _ => "_instance".isPrefixOf s
3312   | _              => false
3313
3314 /--
3315   Sort the given list of `usedParams` using the following order:
3316   - If it is an explicit level `allUserParams`, then use user given order.
3317   - Otherwise, use lexicographical.
3318
3319   Remark: `scopeParams` are the universe params introduced using the `universe` command. `allUserParams` contains
3320   the universe params introduced using the `universe` command *and* the `{...}` notation.
3321
3322   Remark: this function return an exception if there is an `u` not in `usedParams`, that is in `allUserParams` but not in `scopeParams`.
3323
3324   Remark: `explicitParams` are in reverse declaration order. That is, the head is the last declared parameter. -/
3325 def sortDeclLevelParams (scopeParams : List Name) (allUserParams : List Name) (usedParams : Array Name) : Except String (List Name) :=
3326   match allUserParams.find? $ fun u => !usedParams.contains u && !scopeParams.elem u with
3327   | some u => throw s!"unused universe parameter '{u}'"
3328   | none   =>
3329     let result := allUserParams.foldl (fun result levelName => if usedParams.elem levelName then levelName :: result else result) []
3330     let remaining := usedParams.filter (fun levelParam => !allUserParams.elem levelParam)
3331     let remaining := remaining.qsort Name.lt
3332     pure $ result ++ remaining.toList
3333
3334 end Lean.Elab
3335 ::::::::::::::
3336 Elab/DefView.lean

```

```

3337 ::::::::::::::
3338 /-
3339 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
3340 Released under Apache 2.0 license as described in the file LICENSE.
3341 Authors: Leonardo de Moura, Sebastian Ullrich
3342 -/
3343 import Std.ShareCommon
3344 import Lean.Parser.Command
3345 import Lean.Util.CollectLevelParams
3346 import Lean.Util.FoldConsts
3347 import Lean.Meta.CollectFVars
3348 import Lean.Elab.Command
3349 import Lean.Elab.SyntheticMVars
3350 import Lean.Elab.Binders
3351 import Lean.Elab.DeclUtil
3352 namespace Lean.Elab
3353
3354 inductive DefKind where
3355   | «def» | «theorem» | «example» | «opaque» | «abbrev»
3356   deriving Inhabited
3357
3358 def DefKind.isTheorem : DefKind → Bool
3359   | «theorem» => true
3360   | _         => false
3361
3362 def DefKind.isDefOrAbbrevOrOpaque : DefKind → Bool
3363   | «def»      => true
3364   | «opaque»   => true
3365   | «abbrev»   => true
3366   | _         => false
3367
3368 def DefKind.isExample : DefKind → Bool
3369   | «example» => true
3370   | _         => false
3371
3372 structure DefView where
3373   kind      : DefKind
3374   ref       : Syntax
3375   modifiers : Modifiers
3376   declId    : Syntax
3377   binders   : Syntax
3378   type?     : Option Syntax
3379   value     : Syntax
3380   deriving Inhabited
3381
3382 namespace Command
3383

```



```

3384 open Meta
3385
3386 def mkDefViewOfAbbrev (modifiers : Modifiers) (stx : Syntax) : DefView :=
3387   -- leading_parser "abbrev " >> declId >> optDeclSig >> declVal
3388   let (binders, type) := expandOptDeclSig (stx.getArg 2)
3389   let modifiers       := modifiers.addAttribute { name := `inline }
3390   let modifiers       := modifiers.addAttribute { name := `reducible }
3391   { ref := stx, kind := DefKind.abbrev, modifiers := modifiers,
3392     declId := stx.getArg 1, binders := binders, type? := type, value := stx.getArg 3 }
3393
3394 def mkDefViewOfDef (modifiers : Modifiers) (stx : Syntax) : DefView :=
3395   -- leading_parser "def " >> declId >> optDeclSig >> declVal
3396   let (binders, type) := expandOptDeclSig (stx.getArg 2)
3397   { ref := stx, kind := DefKind.def, modifiers := modifiers,
3398     declId := stx.getArg 1, binders := binders, type? := type, value := stx.getArg 3 }
3399
3400 def mkDefViewOfTheorem (modifiers : Modifiers) (stx : Syntax) : DefView :=
3401   -- leading_parser "theorem " >> declId >> declSig >> declVal
3402   let (binders, type) := expandDeclSig (stx.getArg 2)
3403   { ref := stx, kind := DefKind.theorem, modifiers := modifiers,
3404     declId := stx.getArg 1, binders := binders, type? := some type, value := stx.getArg 3 }
3405
3406 namespace MkInstanceName
3407
3408 -- Table for `mkInstanceName`
3409 private def kindReplacements : NameMap String :=
3410   Std.RBMap.ofList [
3411     (`Parser.Term.depArrow, "DepArrow"),
3412     (`Parser.Term.«forall», "Forall"),
3413     (`Parser.Term.arrow, "Arrow"),
3414     (`Parser.Term.prop, "Prop"),
3415     (`Parser.Term.sort, "Sort"),
3416     (`Parser.Term.type, "Type")
3417   ]
3418
3419 abbrev M := StateRefT String CommandElabM
3420
3421 def isFirst : M Bool :=
3422   return (← get) == ""
3423
3424 def append (str : String) : M Unit :=
3425   modify fun s => s ++ str
3426
3427 partial def collect (stx : Syntax) : M Unit := do
3428   match stx with
3429   | Syntax.node k args =>
3430     unless (← isFirst) do

```

```

3431     match kindReplacements.find? k with
3432     | some r => append r
3433     | none   => pure ()
3434   for arg in args do
3435     collect arg
3436 | Syntax.ident (preresolved := preresolved) .. =>
3437   unless preresolved.isEmpty && (← resolveGlobalName stx.getId).isEmpty do
3438     match stx.getId.eraseMacroScopes with
3439     | Name.str _ str _ =>
3440       if str[0].isLower then
3441         append str.capitalize
3442       else
3443         append str
3444     | _ => pure ()
3445 | _ => pure ()
3446
3447 def mkFreshInstanceName : CommandElabM Name := do
3448   let s ← get
3449   let idx := s.nextInstIdx
3450   modify fun s => { s with nextInstIdx := s.nextInstIdx + 1 }
3451   return Lean.Elab.mkFreshInstanceName s.env idx
3452
3453 partial def main (type : Syntax) : CommandElabM Name := do
3454   /- We use `expandMacros` to expand notation such as `x < y` into `LT.lt x y` -/
3455   let type ← liftMacroM <| expandMacros type
3456   let (_, str) ← collect type |>.run ""
3457   if str.isEmpty then
3458     mkFreshInstanceName
3459   else
3460     mkUnusedBaseName <| Name.mkSimple ("inst" ++ str)
3461
3462 end MkInstanceName
3463
3464 def mkDefViewOfConstant (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView := do
3465   -- leading_parser "constant " >> declId >> declSig >> optional declValSimple
3466   let (binders, type) := expandDeclSig (stx.getArg 2)
3467   let val ← match (stx.getArg 3).getOptional? with
3468     | some val => pure val
3469     | none     =>
3470       let val ← `(arbitrary)
3471       pure $ Syntax.node ``Parser.Command.declValSimple #[ mkAtomFrom stx ":", val ]
3472   return {
3473     ref := stx, kind := DefKind.opaque, modifiers := modifiers,
3474     declId := stx.getArg 1, binders := binders, type? := some type, value := val
3475   }
3476
3477 def mkDefViewOfInstance (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView := do

```

```

3478 -- leading_parser Term.attrKind >> "instance " >> optNamedPrio >> optional declId >> declSig >> declVal
3479 let attrKind      ← toAttributeKind stx[0]
3480 let prio          ← liftMacroM <| expandOptNamedPrio stx[2]
3481 let attrStx       ← `(attr| instance $(quote prio):numLit)
3482 let (binders, type) := expandDeclSig stx[4]
3483 let modifiers      := modifiers.addAttribute { kind := attrKind, name := `instance, stx := attrStx }
3484 let declId ← match stx[3].getOptional? with
3485   | some declId => pure declId
3486   | none       =>
3487     let id ← MkInstanceName.main type
3488     pure <| Syntax.node ``Parser.Command.declId #[mkIdentFrom stx id, mkNullNode]
3489 return {
3490   ref := stx, kind := DefKind.def, modifiers := modifiers,
3491   declId := declId, binders := binders, type? := type, value := stx[5]
3492 }
3493
3494 def mkDefViewOfExample (modifiers : Modifiers) (stx : Syntax) : DefView :=
3495   -- leading_parser "example " >> declSig >> declVal
3496   let (binders, type) := expandDeclSig (stx.getArg 1)
3497   let id              := mkIdentFrom stx `example
3498   let declId          := Syntax.node ``Parser.Command.declId #[id, mkNullNode]
3499   { ref := stx, kind := DefKind.example, modifiers := modifiers,
3500     declId := declId, binders := binders, type? := some type, value := stx.getArg 2 }
3501
3502 def isDefLike (stx : Syntax) : Bool :=
3503   let declKind := stx.getKind
3504   declKind == ``Parser.Command.«abbrev» ||
3505   declKind == ``Parser.Command.«def» ||
3506   declKind == ``Parser.Command.«theorem» ||
3507   declKind == ``Parser.Command.«constant» ||
3508   declKind == ``Parser.Command.«instance» ||
3509   declKind == ``Parser.Command.«example»
3510
3511 def mkDefView (modifiers : Modifiers) (stx : Syntax) : CommandElabM DefView :=
3512   let declKind := stx.getKind
3513   if declKind == ``Parser.Command.«abbrev» then
3514     pure $ mkDefViewOfAbbrev modifiers stx
3515   else if declKind == ``Parser.Command.«def» then
3516     pure $ mkDefViewOfDef modifiers stx
3517   else if declKind == ``Parser.Command.«theorem» then
3518     pure $ mkDefViewOfTheorem modifiers stx
3519   else if declKind == ``Parser.Command.«constant» then
3520     mkDefViewOfConstant modifiers stx
3521   else if declKind == ``Parser.Command.«instance» then
3522     mkDefViewOfInstance modifiers stx
3523   else if declKind == ``Parser.Command.«example» then
3524     pure $ mkDefViewOfExample modifiers stx

```

```

3525     else
3526         throwError "unexpected kind of definition"
3527
3528 builtin_initialize registerTraceClass `Elab.definition
3529
3530 end Command
3531 end Lean.Elab
3532 :::::::::::::::
3533 Elab/Deriving.lean
3534 :::::::::::::::
3535 /-
3536 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
3537 Released under Apache 2.0 license as described in the file LICENSE.
3538 Authors: Leonardo de Moura
3539 -/
3540 import Lean.Elab.Deriving.Basic
3541 import Lean.Elab.Deriving.Util
3542 import Lean.Elab.Deriving.Inhabited
3543 import Lean.Elab.Deriving.BEq
3544 import Lean.Elab.Deriving.DecEq
3545 import Lean.Elab.Deriving.Repr
3546 import Lean.Elab.Deriving.FromToJson
3547 import Lean.Elab.Deriving.SizeOf
3548 import Lean.Elab.Deriving.Hashable
3549 import Lean.Elab.Deriving.Ord
3550 :::::::::::::::
3551 Elab/Do.lean
3552 :::::::::::::::
3553 /-
3554 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
3555 Released under Apache 2.0 license as described in the file LICENSE.
3556 Authors: Leonardo de Moura
3557 -/
3558 import Lean.Elab.Term
3559 import Lean.Elab.Binders
3560 import Lean.Elab.Match
3561 import Lean.Elab.Quotation.Util
3562 import Lean.Parser.Do
3563
3564 namespace Lean.Elab.Term
3565 open Lean.Parser.Term
3566 open Meta
3567
3568 private def getDoSeqElems (doSeq : Syntax) : List Syntax :=
3569     if doSeq.getKind == `Lean.Parser.Term.doSeqBracketed then
3570         doSeq[1].getArgs.toList.map fun arg => arg[0]
3571     else if doSeq.getKind == `Lean.Parser.Term.doSeqIndent then

```

```

3572     doSeq[0].getArgs.toList.map fun arg => arg[0]
3573   else
3574     []
3575
3576 private def getDoSeq (doStx : Syntax) : Syntax :=
3577   doStx[1]
3578
3579 @[builtinTermElab liftMethod] def elabLiftMethod : TermElab := fun stx _ =>
3580   throwErrorAt stx "invalid use of `(<- ...)` , must be nested inside a 'do' expression"
3581
3582 /-- Return true if we should not lift `(<- ...)` actions nested in the syntax nodes with the given kind. -/
3583 private def liftMethodDelimiter (k : SyntaxNodeKind) : Bool :=
3584   k == ``Lean.Parser.Term.do ||
3585   k == ``Lean.Parser.Term.doSeqIndent ||
3586   k == ``Lean.Parser.Term.doSeqBracketed ||
3587   k == ``Lean.Parser.Term.termReturn ||
3588   k == ``Lean.Parser.Term.termUnless ||
3589   k == ``Lean.Parser.Term.termTry ||
3590   k == ``Lean.Parser.Term.termFor
3591
3592 /-- Return true if we should generate an error message when lifting a method over this kind of syntax. -/
3593 private def liftMethodForbiddenBinder (k : SyntaxNodeKind) : Bool :=
3594   k == ``Lean.Parser.Term.fun ||
3595   k == ``Lean.Parser.Term.matchAlts
3596
3597 private partial def hasLiftMethod : Syntax → Bool
3598 | Syntax.node k args =>
3599   if liftMethodDelimiter k then false
3600   -- NOTE: We don't check for lifts in quotations here, which doesn't break anything but merely makes this rare case a
3601   -- bit slower
3602   else if k == ``Lean.Parser.Term.liftMethod then true
3603   else args.any hasLiftMethod
3604 | _ => false
3605
3606 structure ExtractMonadResult where
3607   m          : Expr
3608   α          : Expr
3609   hasBindInst : Expr
3610   expectedType : Expr
3611
3612 private def mkIdBindFor (type : Expr) : TermElabM ExtractMonadResult := do
3613   let u ← getDecLevel type
3614   let id      := Lean.mkConst `Id [u]
3615   let idBindVal := Lean.mkConst `Id.hasBind [u]
3616   pure { m := id, hasBindInst := idBindVal, α := type, expectedType := mkApp id type }
3617
3618 private def extractBind (expectedType? : Option Expr) : TermElabM ExtractMonadResult := do

```

```

3619 match expectedType? with
3620 | none => throwError "invalid 'do' notation, expected type is not available"
3621 | some expectedType =>
3622   let type ← withReducible $ whnf expectedType
3623   if type.getAppFn.isMVar then throwError "invalid 'do' notation, expected type is not available"
3624   match type with
3625   | Expr.app m  $\alpha$  _ =>
3626     try
3627       let bindInstType ← mkAppM `Bind #[m]
3628       let bindInstVal ← synthesizeInst bindInstType
3629       pure { m := m, hasBindInst := bindInstVal,  $\alpha$  :=  $\alpha$ , expectedType := expectedType }
3630     catch _ =>
3631       mkIdBindFor type
3632   | _ => mkIdBindFor type
3633
3634 namespace Do
3635
3636 /- A `doMatch` alternative. `vars` is the array of variables declared by `patterns`. -/
3637 structure Alt ( $\sigma$  : Type) where
3638   ref : Syntax
3639   vars : Array Name
3640   patterns : Syntax
3641   rhs :  $\sigma$ 
3642   deriving Inhabited
3643
3644 /-
3645   Auxiliary datastructure for representing a `do` code block, and compiling "reassignments" (e.g., `x := x + 1`).
3646   We convert `Code` into a `Syntax` term representing the:
3647   - `do`-block, or
3648   - the visitor argument for the `forIn` combinator.
3649
3650   We say the following constructors are terminals:
3651   - `break`: for interrupting a `for x in s`
3652   - `continue`: for interrupting the current iteration of a `for x in s`
3653   - `return e`: for returning `e` as the result for the whole `do` computation block
3654   - `action a`: for executing action `a` as a terminal
3655   - `ite`: if-then-else
3656   - `match`: pattern matching
3657   - `jmp` a goto to a join-point
3658
3659   We say the terminals `break`, `continue`, `action`, and `return` are "exit points"
3660
3661   Note that, `return e` is not equivalent to `action (pure e)`. Here is an example:
3662   ```
3663   def f (x : Nat) : IO Unit := do
3664     if x == 0 then
3665       return ()

```

```

3666 IO.println "hello"
3667 ```
3668 Executing `#eval f 0` will not print "hello". Now, consider
3669 ```
3670 def g (x : Nat) : IO Unit := do
3671   if x == 0 then
3672     pure ()
3673   IO.println "hello"
3674 ```
3675 The `if` statement is essentially a noop, and "hello" is printed when we execute `g 0`.
3676
3677 - `decl` represents all declaration-like `doElem`s (e.g., `let`, `have`, `let rec`).
3678   The field `stx` is the actual `doElem`,
3679   `vars` is the array of variables declared by it, and `cont` is the next instruction in the `do` code block.
3680   `vars` is an array since we have declarations such as `let (a, b) := s`.
3681
3682 - `reassign` is an reassignment-like `doElem` (e.g., `x := x + 1`).
3683
3684 - `joinpoint` is a join point declaration: an auxiliary `let`-declaration used to represent the control-flow.
3685
3686 - `seq a k` executes action `a`, ignores its result, and then executes `k`.
3687   We also store the do-elements `dbg_trace` and `assert!` as actions in a `seq`.
3688
3689 A code block `C` is well-formed if
3690 - 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
3691   `ps.size == as.size` -/
3692 inductive Code where
3693 | decl      (xs : Array Name) (doElem : Syntax) (k : Code)
3694 | reassign  (xs : Array Name) (doElem : Syntax) (k : Code)
3695 /- The Boolean value in `params` indicates whether we should use `(x : typeof! x)` when generating term Syntax or not -/
3696 | joinpoint (name : Name) (params : Array (Name × Bool)) (body : Code) (k : Code)
3697 | seq       (action : Syntax) (k : Code)
3698 | action    (action : Syntax)
3699 | «break»   (ref : Syntax)
3700 | «continue» (ref : Syntax)
3701 | «return»  (ref : Syntax) (val : Syntax)
3702 /- Recall that an if-then-else may declare a variable using `optIdent` for the branches `thenBranch` and `elseBranch`. We store the va
3703 | ite       (ref : Syntax) (h? : Option Name) (optIdent : Syntax) (cond : Syntax) (thenBranch : Code) (elseBranch : Code)
3704 | «match»   (ref : Syntax) (gen : Syntax) (discrs : Syntax) (optType : Syntax) (alts : Array (Alt Code))
3705 | jmp       (ref : Syntax) (jpName : Name) (args : Array Syntax)
3706 deriving Inhabited
3707
3708 /- A code block, and the collection of variables updated by it. -/
3709 structure CodeBlock where
3710   code : Code
3711   uvars : NameSet := {} -- set of variables updated by `code`
3712

```

```

3713 private def nameSetToArray (s : NameSet) : Array Name :=
3714   s.fold (fun (xs : Array Name) x => xs.push x) #[]
3715
3716 private def varsToMessageData (vars : Array Name) : MessageData :=
3717   MessageData.joinSep (vars.toList.map fun n => MessageData.ofName (n.simpMacroScopes)) " "
3718
3719 partial def CodeBlock.toMessageData (codeBlock : CodeBlock) : MessageData :=
3720   let us := MessageData.ofList $ (nameSetToArray codeBlock.uvars).toList.map MessageData.ofName
3721   let rec loop : Code → MessageData
3722     | Code.decl xs _ k      => m!"let {varsToMessageData xs} := ...\\n{loop k}"
3723     | Code.reassign xs _ k  => m!"{varsToMessageData xs} := ...\\n{loop k}"
3724     | Code.joinpoint n ps body k => m!"let {n.simpMacroScopes} {varsToMessageData (ps.map Prod.fst)} := {indentD (loop body)}\\n{loop k}"
3725     | Code.seq e k          => m!"{e}\\n{loop k}"
3726     | Code.action e         => e
3727     | Code.ite _ _ _ c t e   => m!"if {c} then {indentD (loop t)}\\nelse{loop e}"
3728     | Code.jump _ j xs      => m!"jmp {j.simpMacroScopes} {xs.toList}"
3729     | Code.«break» _        => m!"break {us}"
3730     | Code.«continue» _     => m!"continue {us}"
3731     | Code.«return» _ v     => m!"return {v} {us}"
3732     | Code.«match» _ _ ds t alts =>
3733       m!"match {ds} with"
3734       ++ alts.foldl (init := m!"") fun acc alt => acc ++ m!"\\n| {alt.patterns} => {loop alt.rhs}"
3735   loop codeBlock.code
3736
3737 /- Return true if the give code contains an exit point that satisfies `p` -/
3738 @[inline] partial def hasExitPointPred (c : Code) (p : Code → Bool) : Bool :=
3739   let rec @[specialize] loop : Code → Bool
3740     | Code.decl _ _ k      => loop k
3741     | Code.reassign _ _ k  => loop k
3742     | Code.joinpoint _ _ b k  => loop b || loop k
3743     | Code.seq _ k         => loop k
3744     | Code.ite _ _ _ t e    => loop t || loop e
3745     | Code.«match» _ _ _ alts => alts.any (loop .rhs)
3746     | Code.jump _ _ _      => false
3747     | c                    => p c
3748   loop c
3749
3750 def hasExitPoint (c : Code) : Bool :=
3751   hasExitPointPred c fun c => true
3752
3753 def hasReturn (c : Code) : Bool :=
3754   hasExitPointPred c fun
3755     | Code.«return» _ _ => true
3756     | _                 => false
3757
3758 def hasTerminalAction (c : Code) : Bool :=
3759   hasExitPointPred c fun

```



```

3760 | Code.«action» _ => true
3761 | _ => false
3762
3763 def hasBreakContinue (c : Code) : Bool :=
3764   hasExitPointPred c fun
3765     | Code.«break» _      => true
3766     | Code.«continue» _  => true
3767     | _                  => false
3768
3769 def hasBreakContinueReturn (c : Code) : Bool :=
3770   hasExitPointPred c fun
3771     | Code.«break» _      => true
3772     | Code.«continue» _  => true
3773     | Code.«return» _ _  => true
3774     | _                  => false
3775
3776 def mkAuxDeclFor {m} [Monad m] [MonadQuotation m] (e : Syntax) (mkCont : Syntax → m Code) : m Code := withRef e <| withFreshMacroScope d
3777   let y ← `(y)
3778   let yName := y.getId
3779   let doElem ← `(doElem| let y ← $e:term)
3780   -- Add elaboration hint for producing sane error message
3781   let y ← `(ensureExpectedType% "type mismatch, result value" $y)
3782   let k ← mkCont y
3783   pure $ Code.decl #[yName] doElem k
3784
3785 /- Convert `action _ e` instructions in `c` into `let y ← e; jmp _ jp (xs y)` -/
3786 partial def convertTerminalActionIntoJmp (code : Code) (jp : Name) (xs : Array Name) : MacroM Code :=
3787   let rec loop : Code → MacroM Code
3788     | Code.decl xs stx k      => do Code.decl xs stx (← loop k)
3789     | Code.reassign xs stx k => do Code.reassign xs stx (← loop k)
3790     | Code.joinpoint n ps b k => do Code.joinpoint n ps (← loop b) (← loop k)
3791     | Code.seq e k           => do Code.seq e (← loop k)
3792     | Code.ite ref x? h c t e => do Code.ite ref x? h c (← loop t) (← loop e)
3793     | 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) })
3794     | Code.action e          => mkAuxDeclFor e fun y =>
3795       let ref := e
3796       -- We jump to `jp` with xs **and** y
3797       let jmpArgs := xs.map $ mkIdentFrom ref
3798       let jmpArgs := jmpArgs.push y
3799       pure $ Code.jmp ref jp jmpArgs
3800   | c      => pure c
3801   loop code
3802
3803 structure JPDecl where
3804   name : Name
3805   params : Array (Name × Bool)
3806   body : Code

```

```

3807
3808 def attachJP (jpDecl : JPDecl) (k : Code) : Code :=
3809   Code.joinpoint jpDecl.name jpDecl.params jpDecl.body k
3810
3811 def attachJPs (jpDecls : Array JPDecl) (k : Code) : Code :=
3812   jpDecls.foldr attachJP k
3813
3814 def mkFreshJP (ps : Array (Name × Bool)) (body : Code) : TermElabM JPDecl := do
3815   let ps ←
3816     if ps.isEmpty then
3817       let y ← mkFreshUserName `y
3818       pure #[(y, false)]
3819     else
3820       pure ps
3821   -- Remark: the compiler frontend implemented in C++ currently detects jointpoints created by
3822   -- the "do" notation by testing the name. See hack at method `visit_let` at `lcnf.cpp`
3823   -- We will remove this hack when we re-implement the compiler frontend in Lean.
3824   let name ← mkFreshUserName `_do_jp
3825   pure { name := name, params := ps, body := body }
3826
3827 def mkFreshJP' (xs : Array Name) (body : Code) : TermElabM JPDecl :=
3828   mkFreshJP (xs.map fun x => (x, true)) body
3829
3830 def addFreshJP (ps : Array (Name × Bool)) (body : Code) : StateRefT (Array JPDecl) TermElabM Name := do
3831   let jp ← mkFreshJP ps body
3832   modify fun (jps : Array JPDecl) => jps.push jp
3833   pure jp.name
3834
3835 def insertVars (rs : NameSet) (xs : Array Name) : NameSet :=
3836   xs.foldl (·.insert ·) rs
3837
3838 def eraseVars (rs : NameSet) (xs : Array Name) : NameSet :=
3839   xs.foldl (·.erase ·) rs
3840
3841 def eraseOptVar (rs : NameSet) (x? : Option Name) : NameSet :=
3842   match x? with
3843   | none    => rs
3844   | some x => rs.insert x
3845
3846 /- Create a new jointpoint for `c`, and jump to it with the variables `rs` -/
3847 def mkSimpleJmp (ref : Syntax) (rs : NameSet) (c : Code) : StateRefT (Array JPDecl) TermElabM Code := do
3848   let xs := nameSetToArray rs
3849   let jp ← addFreshJP (xs.map fun x => (x, true)) c
3850   if xs.isEmpty then
3851     let unit ← `(Unit.unit)
3852     return Code.jmp ref jp #[unit]
3853   else

```

```

3854     return Code.jump ref jp (xs.map $ mkIdentFrom ref)
3855
3856 /- Create a new joinpoint that takes `rs` and `val` as arguments. `val` must be syntax representing a pure value.
3857 The body of the joinpoint is created using `mkJPBody yFresh`, where `yFresh`
3858 is a fresh variable created by this method. -/
3859 def mkJmp (ref : Syntax) (rs : NameSet) (val : Syntax) (mkJPBody : Syntax → MacroM Code) : StateRefT (Array JPDecl) TermElabM Code := do
3860   let xs := nameSetToArray rs
3861   let args := xs.map $ mkIdentFrom ref
3862   let args := args.push val
3863   let yFresh ← mkFreshUserName `y
3864   let ps := xs.map fun x => (x, true)
3865   let ps := ps.push (yFresh, false)
3866   let jpBody ← liftMacroM $ mkJPBody (mkIdentFrom ref yFresh)
3867   let jp ← addFreshJP ps jpBody
3868   pure $ Code.jump ref jp args
3869
3870 /- `pullExitPointsAux rs c` auxiliary method for `pullExitPoints`, `rs` is the set of update variable in the current path. -/
3871 partial def pullExitPointsAux : NameSet → Code → StateRefT (Array JPDecl) TermElabM Code
3872 | rs, Code.decl xs stx k          => do Code.decl xs stx (← pullExitPointsAux (eraseVars rs xs) k)
3873 | rs, Code.reassign xs stx k     => do Code.reassign xs stx (← pullExitPointsAux (insertVars rs xs) k)
3874 | rs, Code.joinpoint j ps b k    => do Code.joinpoint j ps (← pullExitPointsAux rs b) (← pullExitPointsAux rs k)
3875 | rs, Code.seq e k               => do Code.seq e (← pullExitPointsAux rs k)
3876 | rs, Code.ite ref x? o c t e    => do Code.ite ref x? o c (← pullExitPointsAux (eraseOptVar rs x?) t) (← pullExitPointsAux (eraseOptVar rs x?) e)
3877 | rs, Code.«match» ref g ds t alts => do
3878   Code.«match» ref g ds t (← alts.mapM fun alt => do pure { alt with rhs := (← pullExitPointsAux (eraseVars rs alt.vars) alt.rhs) })
3879 | rs, c@(Code.jump _ _ _)        => pure c
3880 | rs, Code.«break» ref           => mkSimpleJmp ref rs (Code.«break» ref)
3881 | rs, Code.«continue» ref        => mkSimpleJmp ref rs (Code.«continue» ref)
3882 | rs, Code.«return» ref val      => mkJmp ref rs val (fun y => pure $ Code.«return» ref y)
3883 | rs, Code.action e              =>
3884   -- We use `mkAuxDeclFor` because `e` is not pure.
3885   mkAuxDeclFor e fun y =>
3886     let ref := e
3887     mkJmp ref rs y (fun yFresh => do pure $ Code.action (← `(Pure.pure $yFresh)))
3888
3889 /-
3890 Auxiliary operation for adding new variables to the collection of updated variables in a CodeBlock.
3891 When a new variable is not already in the collection, but is shadowed by some declaration in `c`,
3892 we create auxiliary join points to make sure we preserve the semantics of the code block.
3893 Example: suppose we have the code block `print x; let x := 10; return x`. And we want to extend it
3894 with the reassignment `x := x + 1`. We first use `pullExitPoints` to create
3895 ``
3896 let jp (x!1) := return x!1;
3897 print x;
3898 let x := 10;
3899 jmp jp x
3900 ``

```

```

3901 and then we add the reassignment
3902 ``
3903 x := x + 1
3904 let jp (x!1) := return x!1;
3905 print x;
3906 let x := 10;
3907 jmp jp x
3908 ``
3909 Note that we created a fresh variable `x!1` to avoid accidental name capture.
3910 As another example, consider
3911 ``
3912 print x;
3913 let x := 10
3914 y := y + 1;
3915 return x;
3916 ``
3917 We transform it into
3918 ``
3919 let jp (y x!1) := return x!1;
3920 print x;
3921 let x := 10
3922 y := y + 1;
3923 jmp jp y x
3924 ``
3925 and then we add the reassignment as in the previous example.
3926 We need to include `y` in the jump, because each exit point is implicitly returning the set of
3927 update variables.
3928
3929 We implement the method as follows. Let `us` be `c.uvars`, then
3930 1- for each `return _ y` in `c`, we create a join point
3931   `let j (us y!1) := return y!1`
3932   and replace the `return _ y` with `jmp us y`
3933 2- for each `break`, we create a join point
3934   `let j (us) := break`
3935   and replace the `break` with `jmp us`.
3936 3- Same as 2 for `continue`.
3937 -/
3938 def pullExitPoints (c : Code) : TermElabM Code := do
3939   if hasExitPoint c then
3940     let (c, jpDecls) ← (pullExitPointsAux {} c).run #[]
3941     pure $ attachJPs jpDecls c
3942   else
3943     pure c
3944
3945 partial def extendUpdatedVarsAux (c : Code) (ws : NameSet) : TermElabM Code :=
3946   let rec update : Code → TermElabM Code
3947     | Code.joinpoint j ps b k => do Code.joinpoint j ps (← update b) (← update k)

```

```

3948 | Code.seq e k                      => do Code.seq e (← update k)
3949 | c@(Code.«match» ref g ds t alts) => do
3950   if alts.any fun alt => alt.vars.any fun x => ws.contains x then
3951     -- If a pattern variable is shadowing a variable in ws, we `pullExitPoints`
3952     pullExitPoints c
3953   else
3954     Code.«match» ref g ds t (← alts.mapM fun alt => do pure { alt with rhs := (← update alt.rhs) })
3955 | Code.ite ref none o c t e => do Code.ite ref none o c (← update t) (← update e)
3956 | c@(Code.ite ref (some h) o cond t e) => do
3957   if ws.contains h then
3958     -- if the `h` at `if h:c then t else e` shadows a variable in `ws`, we `pullExitPoints`
3959     pullExitPoints c
3960   else
3961     Code.ite ref (some h) o cond (← update t) (← update e)
3962 | Code.reassign xs stx k => do Code.reassign xs stx (← update k)
3963 | c@(Code.decl xs stx k) => do
3964   if xs.any fun x => ws.contains x then
3965     -- One the declared variables is shadowing a variable in `ws`
3966     pullExitPoints c
3967   else
3968     Code.decl xs stx (← update k)
3969 | c => pure c
3970 update c
3971
3972 /-
3973 Extend the set of updated variables. It assumes `ws` is a super set of `c.uvars`.
3974 We **cannot** simply update the field `c.uvars`, because `c` may have shadowed some variable in `ws`.
3975 See discussion at `pullExitPoints`.
3976 -/
3977 partial def extendUpdatedVars (c : CodeBlock) (ws : NameSet) : TermElabM CodeBlock := do
3978   if ws.any fun x => !c.uvars.contains x then
3979     -- `ws` contains a variable that is not in `c.uvars`, but in `c.dvars` (i.e., it has been shadowed)
3980     pure { code := (← extendUpdatedVarsAux c.code ws), uvars := ws }
3981   else
3982     pure { c with uvars := ws }
3983
3984 private def union (s₁ s₂ : NameSet) : NameSet :=
3985   s₁.fold (·.insert ·) s₂
3986
3987 /-
3988 Given two code blocks `c₁` and `c₂`, make sure they have the same set of updated variables.
3989 Let `ws` the union of the updated variables in `c₁` and `c₂`.
3990 We use `extendUpdatedVars c₁ ws` and `extendUpdatedVars c₂ ws`
3991 -/
3992 def homogenize (c₁ c₂ : CodeBlock) : TermElabM (CodeBlock × CodeBlock) := do
3993   let ws := union c₁.uvars c₂.uvars
3994   let c₁ ← extendUpdatedVars c₁ ws

```

```

3995 let c2 ← extendUpdatedVars c2 ws
3996 pure (c1, c2)
3997
3998 /-
3999 Extending code blocks with variable declarations: `let x : t := v` and `let x : t ← v`.
4000 We remove `x` from the collection of updated variables.
4001 Remark: `stx` is the syntax for the declaration (e.g., `letDecl`), and `xs` are the variables
4002 declared by it. It is an array because we have let-declarations that declare multiple variables.
4003 Example: `let (x, y) := t`
4004 -/
4005 def mkVarDeclCore (xs : Array Name) (stx : Syntax) (c : CodeBlock) : CodeBlock := {
4006   code := Code.decl xs stx c.code,
4007   uvars := eraseVars c.uvars xs
4008 }
4009
4010 /-
4011 Extending code blocks with reassignments: `x : t := v` and `x : t ← v`.
4012 Remark: `stx` is the syntax for the declaration (e.g., `letDecl`), and `xs` are the variables
4013 declared by it. It is an array because we have let-declarations that declare multiple variables.
4014 Example: `(x, y) ← t`
4015 -/
4016 def mkReassignCore (xs : Array Name) (stx : Syntax) (c : CodeBlock) : TermElabM CodeBlock := do
4017   let us := c.uvars
4018   let ws := insertVars us xs
4019   -- If `xs` contains a new updated variable, then we must use `extendUpdatedVars`.
4020   -- See discussion at `pullExitPoints`
4021   let code ← if xs.any fun x => !us.contains x then extendUpdatedVarsAux c.code ws else pure c.code
4022   pure { code := Code.reassign xs stx code, uvars := ws }
4023
4024 def mkSeq (action : Syntax) (c : CodeBlock) : CodeBlock :=
4025   { c with code := Code.seq action c.code }
4026
4027 def mkTerminalAction (action : Syntax) : CodeBlock :=
4028   { code := Code.action action }
4029
4030 def mkReturn (ref : Syntax) (val : Syntax) : CodeBlock :=
4031   { code := Code.«return» ref val }
4032
4033 def mkBreak (ref : Syntax) : CodeBlock :=
4034   { code := Code.«break» ref }
4035
4036 def mkContinue (ref : Syntax) : CodeBlock :=
4037   { code := Code.«continue» ref }
4038
4039 def mkIte (ref : Syntax) (optIdent : Syntax) (cond : Syntax) (thenBranch : CodeBlock) (elseBranch : CodeBlock) : TermElabM CodeBlock := do
4040   let x? := if optIdent.isNone then none else some optIdent[0].getId
4041   let (thenBranch, elseBranch) ← homogenize thenBranch elseBranch

```

```

4042 pure {
4043   code := Code.ite ref x? optIdent cond thenBranch.code elseBranch.code,
4044   uvars := thenBranch.uvars,
4045 }
4046
4047 private def mkUnit : MacroM Syntax :=
4048   `(({}) : PUnit))
4049
4050 private def mkPureUnit : MacroM Syntax :=
4051   `(pure PUnit.unit)
4052
4053 def mkPureUnitAction : MacroM CodeBlock := do
4054   mkTerminalAction (← mkPureUnit)
4055
4056 def mkUnless (cond : Syntax) (c : CodeBlock) : MacroM CodeBlock := do
4057   let thenBranch ← mkPureUnitAction
4058   pure { c with code := Code.ite (← getRef) none mkNullNode cond thenBranch.code c.code }
4059
4060 def mkMatch (ref : Syntax) (genParam : Syntax) (discrs : Syntax) (optType : Syntax) (alts : Array (Alt CodeBlock)) : TermElabM CodeBlock
4061   -- nary version of homogenize
4062   let ws := alts.foldl (union · .rhs.uvars) {}
4063   let alts ← alts.mapM fun alt => do
4064     let rhs ← extendUpdatedVars alt.rhs ws
4065     pure { ref := alt.ref, vars := alt.vars, patterns := alt.patterns, rhs := rhs.code : Alt Code }
4066   pure { code := Code.«match» ref genParam discrs optType alts, uvars := ws }
4067
4068 /- Return a code block that executes `terminal` and then `k` with the value produced by `terminal`.
4069 This method assumes `terminal` is a terminal -/
4070 def concat (terminal : CodeBlock) (kRef : Syntax) (y? : Option Name) (k : CodeBlock) : TermElabM CodeBlock := do
4071   unless hasTerminalAction terminal.code do
4072     throwErrorAt kRef "'do' element is unreachable"
4073   let (terminal, k) ← homogenize terminal k
4074   let xs := nameSetToArray k.uvars
4075   let y ← match y? with | some y => pure y | none => mkFreshUserName `y
4076   let ps := xs.map fun x => (x, true)
4077   let ps := ps.push (y, false)
4078   let jpDecl ← mkFreshJP ps k.code
4079   let jp := jpDecl.name
4080   let terminal ← liftMacroM $ convertTerminalActionIntoJump terminal.code jp xs
4081   pure { code := attachJP jpDecl terminal, uvars := k.uvars }
4082
4083 def getLetIdDeclVar (letIdDecl : Syntax) : Name :=
4084   letIdDecl[0].getId
4085
4086 def getPatternVarNames (pvars : Array PatternVar) : Array Name :=
4087   pvars.filterMap fun
4088     | PatternVar.localVar x => some x

```

```

4089 | _ => none
4090
4091 -- support both regular and syntax match
4092 def getPatternVarsEx (pattern : Syntax) : TermElabM (Array Name) :=
4093   getPatternVarNames <$> getPatternVars pattern <|>
4094   Array.map Syntax.getId <$> Quotation.getPatternVars pattern
4095
4096 def getPatternsVarsEx (patterns : Array Syntax) : TermElabM (Array Name) :=
4097   getPatternVarNames <$> getPatternsVars patterns <|>
4098   Array.map Syntax.getId <$> Quotation.getPatternsVars patterns
4099
4100 def getLetPatDeclVars (letPatDecl : Syntax) : TermElabM (Array Name) := do
4101   let pattern := letPatDecl[0]
4102   getPatternVarsEx pattern
4103
4104 def getLetEqnsDeclVar (letEqnsDecl : Syntax) : Name :=
4105   letEqnsDecl[0].getId
4106
4107 def getLetDeclVars (letDecl : Syntax) : TermElabM (Array Name) := do
4108   let arg := letDecl[0]
4109   if arg.getKind == `Lean.Parser.Term.letIdDecl then
4110     pure #[getLetIdDeclVar arg]
4111   else if arg.getKind == `Lean.Parser.Term.letPatDecl then
4112     getLetPatDeclVars arg
4113   else if arg.getKind == `Lean.Parser.Term.letEqnsDecl then
4114     pure #[getLetEqnsDeclVar arg]
4115   else
4116     throwError "unexpected kind of let declaration"
4117
4118 def getDoLetVars (doLet : Syntax) : TermElabM (Array Name) :=
4119   -- leading_parser "let " >> optional "mut " >> letDecl
4120   getLetDeclVars doLet[2]
4121
4122 def getDoHaveVar (doHave : Syntax) : Name :=
4123   /-
4124     `leading_parser "have " >> Term.haveDecl`
4125     where
4126     ```
4127     haveDecl := leading_parser optIdent >> termParser >> (haveAssign <|> fromTerm <|> byTactic)
4128     optIdent := optional (try (ident >> " : "))
4129     ```
4130   -/
4131   let optIdent := doHave[1][0]
4132   if optIdent.isNone then
4133     `this
4134   else

```



```

4136     optIdent[0].getId
4137
4138 def getDoLetRecVars (doLetRec : Syntax) : TermElabM (Array Name) := do
4139   -- letRecDecls is an array of `(group (optional attributes >> letDecl))`
4140   let letRecDecls := doLetRec[1][0].getSepArgs
4141   let letDecls := letRecDecls.map fun p => p[2]
4142   let mut allVars := #[]
4143   for letDecl in letDecls do
4144     let vars ← getLetDeclVars letDecl
4145     allVars := allVars ++ vars
4146   pure allVars
4147
4148 -- ident >> optType >> leftArrow >> termParser
4149 def getDoIdDeclVar (doIdDecl : Syntax) : Name :=
4150   doIdDecl[0].getId
4151
4152 -- termParser >> leftArrow >> termParser >> optional (" | " >> termParser)
4153 def getDoPatDeclVars (doPatDecl : Syntax) : TermElabM (Array Name) := do
4154   let pattern := doPatDecl[0]
4155   getPatternVarsEx pattern
4156
4157 -- leading_parser "let " >> optional "mut " >> (doIdDecl <|> doPatDecl)
4158 def getDoLetArrowVars (doLetArrow : Syntax) : TermElabM (Array Name) := do
4159   let decl := doLetArrow[2]
4160   if decl.getKind == `Lean.Parser.Term.doIdDecl then
4161     pure #[getDoIdDeclVar decl]
4162   else if decl.getKind == `Lean.Parser.Term.doPatDecl then
4163     getDoPatDeclVars decl
4164   else
4165     throwError "unexpected kind of 'do' declaration"
4166
4167 def getDoReassignVars (doReassign : Syntax) : TermElabM (Array Name) := do
4168   let arg := doReassign[0]
4169   if arg.getKind == `Lean.Parser.Term.letIdDecl then
4170     pure #[getLetIdDeclVar arg]
4171   else if arg.getKind == `Lean.Parser.Term.letPatDecl then
4172     getLetPatDeclVars arg
4173   else
4174     throwError "unexpected kind of reassignment"
4175
4176 def mkDoSeq (doElems : Array Syntax) : Syntax :=
4177   mkNode `Lean.Parser.Term.doSeqIndent #[mkNullNode $ doElems.map fun doElem => mkNullNode #[doElem, mkNullNode]]
4178
4179 def mkSingletonDoSeq (doElem : Syntax) : Syntax :=
4180   mkDoSeq #[doElem]
4181
4182 /-

```

```

4183   If the given syntax is a `doIf`, return an equivalent `doIf` that has an `else` but no `else if`s or `if let`s. -/
4184 private def expandDoIf? (stx : Syntax) : MacroM (Option Syntax) := match stx with
4185 | `(doElem|if $p:doIfProp then $t else $e) => pure none
4186 | `(doElem|if%$i $cond:doIfCond then $t $[else if%$is $conds:doIfCond then $ts]* $[else $e?]?) => withRef stx do
4187   let mut e      := e?.getD (← `(doSeq|pure PUnit.unit))
4188   let mut eIsSeq := true
4189   for (i, cond, t) in Array.zip (is.reverse.push i) (Array.zip (conds.reverse.push cond) (ts.reverse.push t)) do
4190     e ← if eIsSeq then e else `(doSeq|$e:doElem)
4191     e ← withRef cond <| match cond with
4192     | `(doIfCond|let $pat := $d) => `(doElem| match%$i $d:term with | $pat:term => $t | _ => $e)
4193     | `(doIfCond|let $pat ← $d) => `(doElem| match%$i ← $d with | $pat:term => $t | _ => $e)
4194     | `(doIfCond|$cond:doIfProp) => `(doElem| if%$i $cond:doIfProp then $t else $e)
4195     | _ => `(doElem| if%$i $(Syntax.missing) then $t else $e)
4196   eIsSeq := false
4197   return some e
4198 | _ => pure none
4199
4200 structure DoIfView where
4201   ref      : Syntax
4202   optIdent : Syntax
4203   cond     : Syntax
4204   thenBranch : Syntax
4205   elseBranch : Syntax
4206
4207 /- This method assumes `expandDoIf?` is not applicable. -/
4208 private def mkDoIfView (doIf : Syntax) : MacroM DoIfView := do
4209   pure {
4210     ref      := doIf,
4211     optIdent := doIf[1][0],
4212     cond     := doIf[1][1],
4213     thenBranch := doIf[3],
4214     elseBranch := doIf[5][1]
4215   }
4216
4217 /-
4218 We use `MProd` instead of `Prod` to group values when expanding the
4219 `do` notation. `MProd` is a universe monomorphic product.
4220 The motivation is to generate simpler universe constraints in code
4221 that was not written by the user.
4222 Note that we are not restricting the macro power since the
4223 `Bind.bind` combinator already forces values computed by monadic
4224 actions to be in the same universe.
4225 -/
4226 private def mkTuple (elems : Array Syntax) : MacroM Syntax := do
4227   if elems.size == 0 then
4228     mkUnit
4229   else if elems.size == 1 then

```

```

4230     pure elems[0]
4231   else
4232     (elems.extract 0 (elems.size - 1)).foldrM
4233       (fun elem tuple => `(MProd.mk $elem $tuple))
4234       (elems.back)
4235
4236 /- Return `some action` if `doElem` is a `doExpr <action>` -/
4237 def isDoExpr? (doElem : Syntax) : Option Syntax :=
4238   if doElem.getKind == `Lean.Parser.Term.doExpr then
4239     some doElem[0]
4240   else
4241     none
4242
4243 /--
4244   Given `uvars := #[a_1, ..., a_n, a_{n+1}]` construct term
4245   ```
4246   let a_1      := x.1
4247   let x        := x.2
4248   let a_2      := x.1
4249   let x        := x.2
4250   ...
4251   let a_n      := x.1
4252   let a_{n+1} := x.2
4253   body
4254   ```
4255   Special cases
4256   - `uvars := #[]` => `body`
4257   - `uvars := #[a]` => `let a := x; body`
4258
4259
4260   We use this method when expanding the `for-in` notation.
4261 -/
4262 private def destructTuple (uvars : Array Name) (x : Syntax) (body : Syntax) : MacroM Syntax := do
4263   if uvars.size == 0 then
4264     return body
4265   else if uvars.size == 1 then
4266     `(let $(← mkIdentFromRef uvars[0]):ident := $x; $body)
4267   else
4268     destruct uvars.toList x body
4269 where
4270   destruct (as : List Name) (x : Syntax) (body : Syntax) : MacroM Syntax := do
4271     match as with
4272     | [a, b] => `(let $(← mkIdentFromRef a):ident := $x.1; let $(← mkIdentFromRef b):ident := $x.2; $body)
4273     | a :: as => withFreshMacroScope do
4274       let rest ← destruct as (← `(x)) body
4275       `(let $(← mkIdentFromRef a):ident := $x.1; let x := $x.2; $rest)
4276     | _ => unreachable!

```

4277

4278 /-

4279 The procedure ``ToTerm.run`` converts a ``CodeBlock`` into a ``Syntax`` term.

4280 We use this method to convert

4281 1- The ``CodeBlock`` for a root ``do ...`` term into a ``Syntax`` term. This kind of

4282 ``CodeBlock`` never contains ``break`` nor ``continue``. Moreover, the collection

4283 of updated variables is not packed into the result.

4284 Thus, we have two kinds of exit points

4285 - ``Code.action e`` which is converted into ``e``

4286 - ``Code.return _ e`` which is converted into ``pure e``

4287

4288 We use ``Kind.regular`` for this case.

4289

4290 2- The ``CodeBlock`` for ``b`` at ``for x in xs do b``. In this case, we need to generate

4291 a ``Syntax`` term representing a function for the ``xs.forIn`` combinator.

4292

4293 a) If ``b`` contain a ``Code.return _ a`` exit point. The generated ``Syntax`` term

4294 has type ``m (ForInStep (Option $\alpha \times \sigma$))``, where ``a : $\alpha`$` , and the `` $\sigma`$` is the type

4295 of the tuple of variables reassigned by ``b``.

4296 We use ``Kind.forInWithReturn`` for this case

4297

4298 b) If ``b`` does not contain a ``Code.return _ a`` exit point. Then, the generated

4299 ``Syntax`` term has type ``m (ForInStep σ)``.

4300 We use ``Kind.forIn`` for this case.

4301

4302 3- The ``CodeBlock`` ``c`` for a ``do`` sequence nested in a monadic combinator (e.g., ``MonadExcept.tryCatch``).

4303

4304 The generated ``Syntax`` term for ``c`` must inform whether ``c`` "exited" using ``Code.action``, ``Code.return``,

4305 ``Code.break`` or ``Code.continue``. We use the auxiliary types ``DoResult`s` for storing this information.

4306 For example, the auxiliary type ``DoResultPBC $\alpha \sigma`$` is used for a code block that exits with ``Code.action``,

4307 `**and**` ``Code.break`/`Code.continue``, `` $\alpha`$` is the type of values produced by the exit ``action``, and

4308 `` $\sigma`$` is the type of the tuple of reassigned variables.

4309 The type ``DoResult $\alpha \beta \sigma`$` is used for code blocks that exit with

4310 ``Code.action``, ``Code.return``, `**and**` ``Code.break`/`Code.continue``, `` $\beta`$` is the type of the returned values.

4311 We don't use ``DoResult $\alpha \beta \sigma`$` for all cases because:

4312

4313 a) The elaborator would not be able to infer all type parameters without extra annotations. For example,

4314 if the code block does not contain ``Code.return _ _``, the elaborator will not be able to infer `` $\beta`$` .

4315

4316 b) We need to pattern match on the result produced by the combinator (e.g., ``MonadExcept.tryCatch``),

4317 but we don't want to consider "unreachable" cases.

4318

4319 We do not distinguish between cases that contain ``break``, but not ``continue``, and vice versa.

4320

4321 When listing all cases, we use ``a`` to indicate the code block contains ``Code.action _``, ``r`` for ``Code.return _ _``,

4322 and ``b/c`` for a code block that contains ``Code.break _`` or ``Code.continue _``.

4323

```

4324 - `a`: `Kind.regular`, type `m (α × σ)`
4325
4326 - `r`: `Kind.regular`, type `m (α × σ)`
4327     Note that the code that pattern matches on the result will behave differently in this case.
4328     It produces `return a` for this case, and `pure a` for the previous one.
4329
4330 - `b/c`: `Kind.nestedBC`, type `m (DoResultBC σ)`
4331
4332 - `a` and `r`: `Kind.nestedPR`, type `m (DoResultPR α β σ)`
4333
4334 - `a` and `bc`: `Kind.nestedSBC`, type `m (DoResultSBC α σ)`
4335
4336 - `r` and `bc`: `Kind.nestedSBC`, type `m (DoResultSBC α σ)`
4337     Again the code that pattern matches on the result will behave differently in this case and
4338     the previous one. It produces `return a` for the constructor `DoResultSPR.pureReturn a u` for
4339     this case, and `pure a` for the previous case.
4340
4341 - `a`, `r`, `b/c`: `Kind.nestedPRBC`, type type `m (DoResultPRBC α β σ)`
4342
4343 Here is the recipe for adding new combinators with nested `do`s.
4344 Example: suppose we want to support `repeat doSeq`. Assuming we have `repeat : m α → m α`
4345 1- Convert `doSeq` into `codeBlock : CodeBlock`
4346 2- Create term `term` using `mkNestedTerm code m uvars a r bc` where
4347     `code` is `codeBlock.code`, `uvars` is an array containing `codeBlock.uvars`,
4348     `m` is a `Syntax` representing the Monad, and
4349     `a` is true if `code` contains `Code.action _`,
4350     `r` is true if `code` contains `Code.return _ _`,
4351     `bc` is true if `code` contains `Code.break _` or `Code.continue _`.
4352
4353 Remark: for combinators such as `repeat` that take a single `doSeq`, all
4354 arguments, but `m`, are extracted from `codeBlock`.
4355 3- Create the term `repeat $term`
4356 4- and then, convert it into a `doSeq` using `matchNestedTermResult ref (repeat $term) uvsar a r bc`
4357
4358 -/
4359 namespace ToTerm
4360
4361 inductive Kind where
4362 | regular
4363 | forIn
4364 | forInWithReturn
4365 | nestedBC
4366 | nestedPR
4367 | nestedSBC
4368 | nestedPRBC
4369
4370 instance : Inhabited Kind := (Kind.regular)

```

```

4371
4372 def Kind.isRegular : Kind → Bool
4373 | Kind.regular => true
4374 | _           => false
4375
4376 structure Context where
4377   m      : Syntax -- Syntax to reference the monad associated with the do notation.
4378   uvars  : Array Name
4379   kind   : Kind
4380
4381 abbrev M := ReaderT Context MacroM
4382
4383 def mkUVarTuple : M Syntax := do
4384   let ctx ← read
4385   let uvarIdents ← ctx.uvars.mapM mkIdentFromRef
4386   mkTuple uvarIdents
4387
4388 def returnToTerm (val : Syntax) : M Syntax := do
4389   let ctx ← read
4390   let u ← mkUVarTuple
4391   match ctx.kind with
4392   | Kind.regular      => if ctx.uvars.isEmpty then `(Pure.pure $val) else `(Pure.pure (MProd.mk $val $u))
4393   | Kind.forIn        => `(Pure.pure (ForInStep.done $u))
4394   | Kind.forInWithReturn => `(Pure.pure (ForInStep.done (MProd.mk (some $val) $u)))
4395   | Kind.nestedBC      => unreachable!
4396   | Kind.nestedPR      => `(Pure.pure (DoResultPR.«return» $val $u))
4397   | Kind.nestedSBC     => `(Pure.pure (DoResultSBC.«pureReturn» $val $u))
4398   | Kind.nestedPRBC    => `(Pure.pure (DoResultPRBC.«return» $val $u))
4399
4400 def continueToTerm : M Syntax := do
4401   let ctx ← read
4402   let u ← mkUVarTuple
4403   match ctx.kind with
4404   | Kind.regular      => unreachable!
4405   | Kind.forIn        => `(Pure.pure (ForInStep.yield $u))
4406   | Kind.forInWithReturn => `(Pure.pure (ForInStep.yield (MProd.mk none $u)))
4407   | Kind.nestedBC      => `(Pure.pure (DoResultBC.«continue» $u))
4408   | Kind.nestedPR      => unreachable!
4409   | Kind.nestedSBC     => `(Pure.pure (DoResultSBC.«continue» $u))
4410   | Kind.nestedPRBC    => `(Pure.pure (DoResultPRBC.«continue» $u))
4411
4412 def breakToTerm : M Syntax := do
4413   let ctx ← read
4414   let u ← mkUVarTuple
4415   match ctx.kind with
4416   | Kind.regular      => unreachable!
4417   | Kind.forIn        => `(Pure.pure (ForInStep.done $u))

```

```

4418 | Kind.forInWithReturn => `(Pure.pure (ForInStep.done (MProd.mk none $u)))
4419 | Kind.nestedBC       => `(Pure.pure (DoResultBC.«break» $u))
4420 | Kind.nestedPR       => unreachable!
4421 | Kind.nestedSBC      => `(Pure.pure (DoResultSBC.«break» $u))
4422 | Kind.nestedPRBC     => `(Pure.pure (DoResultPRBC.«break» $u))
4423
4424 def actionTerminalToTerm (action : Syntax) : M Syntax := withRef action <| withFreshMacroScope do
4425   let ctx ← read
4426   let u ← mkUVarTuplet
4427   match ctx.kind with
4428   | Kind.regular      => if ctx.uvars.isEmpty then pure action else `(Bind.bind $action fun y => Pure.pure (MProd.mk y $u))
4429   | Kind.forIn        => `(Bind.bind $action fun (_ : PUnit) => Pure.pure (ForInStep.yield $u))
4430   | Kind.forInWithReturn => `(Bind.bind $action fun (_ : PUnit) => Pure.pure (ForInStep.yield (MProd.mk none $u)))
4431   | Kind.nestedBC     => unreachable!
4432   | Kind.nestedPR     => `(Bind.bind $action fun y => (Pure.pure (DoResultPR.«pure» y $u)))
4433   | Kind.nestedSBC    => `(Bind.bind $action fun y => (Pure.pure (DoResultSBC.«pureReturn» y $u)))
4434   | Kind.nestedPRBC   => `(Bind.bind $action fun y => (Pure.pure (DoResultPRBC.«pure» y $u)))
4435
4436 def seqToTerm (action : Syntax) (k : Syntax) : M Syntax := withRef action <| withFreshMacroScope do
4437   if action.getKind == `Lean.Parser.Term.doDbgTrace then
4438     let msg := action[1]
4439     `(dbg_trace $msg; $k)
4440   else if action.getKind == `Lean.Parser.Term.doAssert then
4441     let cond := action[1]
4442     `(assert! $cond; $k)
4443   else
4444     let action ← withRef action `(($action : $(←read).m) PUnit))
4445     `(Bind.bind $action (fun (_ : PUnit) => $k))
4446
4447 def declToTerm (decl : Syntax) (k : Syntax) : M Syntax := withRef decl <| withFreshMacroScope do
4448   let kind := decl.getKind
4449   if kind == `Lean.Parser.Term.doLet then
4450     let letDecl := decl[2]
4451     `(let $letDecl:letDecl; $k)
4452   else if kind == `Lean.Parser.Term.doLetRec then
4453     let letRecToken := decl[0]
4454     let letRecDecls := decl[1]
4455     pure $ mkNode `Lean.Parser.Term.letrec #[letRecToken, letRecDecls, mkNullNode, k]
4456   else if kind == `Lean.Parser.Term.doLetArrow then
4457     let arg := decl[2]
4458     let ref := arg
4459     if arg.getKind == `Lean.Parser.Term.doIdDecl then
4460       let id := arg[0]
4461       let type := expandOptType ref arg[1]
4462       let doElem := arg[3]
4463       -- `doElem` must be a `doExpr action`. See `doLetArrowToCode`
4464       match isDoExpr? doElem with

```

```

4465 | some action =>
4466   let action ← withRef action `(($action : $(← read).m) $type))
4467   `(Bind.bind $action (fun ($id:ident : $type) => $k))
4468 | none      => Macro.throwErrorAt decl "unexpected kind of 'do' declaration"
4469 else
4470   Macro.throwErrorAt decl "unexpected kind of 'do' declaration"
4471 else if kind == `Lean.Parser.Term.doHave then
4472   -- The `have` term is of the form `"have " >> haveDecl >> optSemicolon termParser`
4473   let args := decl.getArgs
4474   let args := args ++ #[mkNullNode /- optional ';' -/, k]
4475   pure $ mkNode `Lean.Parser.Term.«have» args
4476 else
4477   Macro.throwErrorAt decl "unexpected kind of 'do' declaration"
4478
4479 def reassignToTerm (reassign : Syntax) (k : Syntax) : MacroM Syntax := withRef reassign <| withFreshMacroScope do
4480   let kind := reassign.getKind
4481   if kind == `Lean.Parser.Term.doReassign then
4482     -- doReassign := leading_parser (letIdDecl <|> letPatDecl)
4483     let arg := reassign[0]
4484     if arg.getKind == `Lean.Parser.Term.letIdDecl then
4485       -- letIdDecl := leading_parser ident >> many (ppSpace >> bracketedBinder) >> optType >> " := " >> termParser
4486       let x := arg[0]
4487       let val := arg[4]
4488       let newVal ← `(ensureTypeOf% $x $(quote "invalid reassignment, value") $val)
4489       let arg := arg.setArg 4 newVal
4490       let letDecl := mkNode `Lean.Parser.Term.letDecl #[arg]
4491       `(let $letDecl:letDecl; $k)
4492     else
4493       -- TODO: ensure the types did not change
4494       let letDecl := mkNode `Lean.Parser.Term.letDecl #[arg]
4495       `(let $letDecl:letDecl; $k)
4496   else
4497     -- Note that `doReassignArrow` is expanded by `doReassignArrowToCode`
4498     Macro.throwErrorAt reassign "unexpected kind of 'do' reassignment"
4499
4500 def mkIte (optIdent : Syntax) (cond : Syntax) (thenBranch : Syntax) (elseBranch : Syntax) : MacroM Syntax := do
4501   if optIdent.isNone then
4502     `(ite $cond $thenBranch $elseBranch)
4503   else
4504     let h := optIdent[0]
4505     `(dite $cond (fun $h => $thenBranch) (fun $h => $elseBranch))
4506
4507 def mkJoinPoint (j : Name) (ps : Array (Name × Bool)) (body : Syntax) (k : Syntax) : M Syntax := withRef body <| withFreshMacroScope do
4508   let pTypes ← ps.mapM fun (id, useTypeOf) => do if useTypeOf then `(typeOf% $(← mkIdentFromRef id)) else `(_)
4509   let ps ← ps.mapM fun (id, useTypeOf) => mkIdentFromRef id
4510   /-
4511   We use `let_delayed` instead of `let` for joinpoints to make sure `$k` is elaborated before `$body`.

```



```

4512 By elaborating ` $k ` first, we "learn" more about ` $body ` 's type.
4513 For example, consider the following example `do` expression
4514 ```
4515 def f (x : Nat) : IO Unit := do
4516   if x > 0 then
4517     IO.println "x is not zero" -- Error is here
4518   IO.mkRef true
4519 ```
4520 it is expanded into
4521 ```
4522 def f (x : Nat) : IO Unit := do
4523   let jp (u : Unit) : IO _ :=
4524     IO.mkRef true;
4525   if x > 0 then
4526     IO.println "not zero"
4527     jp ()
4528   else
4529     jp ()
4530 ```
4531 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`.
4532 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 →`
4533 Then, we get the expected type mismatch error at `IO.mkRef true`. -/
4534 `(let_delayed $(← mkIdentFromRef j):ident $[( $ps : $pTypes)]* : $(← read).m) _ := $body; $k)
4535
4536 def mkJmp (ref : Syntax) (j : Name) (args : Array Syntax) : Syntax :=
4537   Syntax.mkApp (mkIdentFrom ref j) args
4538
4539 partial def toTerm : Code → M Syntax
4540 | Code.«return» ref val => withRef ref <| returnToTerm val
4541 | Code.«continue» ref => withRef ref continueToTerm
4542 | Code.«break» ref => withRef ref breakToTerm
4543 | Code.action e => actionTerminalToTerm e
4544 | Code.joinpoint j ps b k => do mkJoinPoint j ps (← toTerm b) (← toTerm k)
4545 | Code.jmp ref j args => pure $ mkJmp ref j args
4546 | Code.decl _ stx k => do declToTerm stx (← toTerm k)
4547 | Code.reassign _ stx k => do reassignToTerm stx (← toTerm k)
4548 | Code.seq stx k => do seqToTerm stx (← toTerm k)
4549 | Code.ite ref _ o c t e => withRef ref <| do mkIte o c (← toTerm t) (← toTerm e)
4550 | Code.«match» ref genParam discrs optType alts => do
4551   let mut termAlts := #[]
4552   for alt in alts do
4553     let rhs ← toTerm alt.rhs
4554     let termAlt := mkNode `Lean.Parser.Term.matchAlt #[mkAtomFrom alt.ref "|", alt.patterns, mkAtomFrom alt.ref "=>", rhs]
4555     termAlts := termAlts.push termAlt
4556   let termMatchAlts := mkNode `Lean.Parser.Term.matchAlts #[mkNullNode termAlts]
4557   pure $ mkNode `Lean.Parser.Term.«match» #[mkAtomFrom ref "match", genParam, discrs, optType, mkAtomFrom ref "with", termMatchAlts]
4558

```

```

4559 def run (code : Code) (m : Syntax) (uvars : Array Name := #[]) (kind := Kind.regular) : MacroM Syntax := do
4560   let term ← toTerm code { m := m, kind := kind, uvars := uvars }
4561   pure term
4562
4563 /- Given
4564   - `a` is true if the code block has a `Code.action` exit point
4565   - `r` is true if the code block has a `Code.return` exit point
4566   - `bc` is true if the code block has a `Code.break` or `Code.continue` exit point
4567
4568   generate Kind. See comment at the beginning of the `ToTerm` namespace. -/
4569 def mkNestedKind (a r bc : Bool) : Kind :=
4570   match a, r, bc with
4571   | true, false, false => Kind.regular
4572   | false, true, false => Kind.regular
4573   | false, false, true  => Kind.nestedBC
4574   | true, true, false   => Kind.nestedPR
4575   | true, false, true   => Kind.nestedSBC
4576   | false, true, true   => Kind.nestedSBC
4577   | true, true, true    => Kind.nestedPRBC
4578   | false, false, false => unreachable!
4579
4580 def mkNestedTerm (code : Code) (m : Syntax) (uvars : Array Name) (a r bc : Bool) : MacroM Syntax := do
4581   ToTerm.run code m uvars (mkNestedKind a r bc)
4582
4583 /- Given a term `term` produced by `ToTerm.run`, pattern match on its result.
4584   See comment at the beginning of the `ToTerm` namespace.
4585
4586   - `a` is true if the code block has a `Code.action` exit point
4587   - `r` is true if the code block has a `Code.return` exit point
4588   - `bc` is true if the code block has a `Code.break` or `Code.continue` exit point
4589
4590   The result is a sequence of `doElem` -/
4591 def matchNestedTermResult (term : Syntax) (uvars : Array Name) (a r bc : Bool) : MacroM (List Syntax) := do
4592   let toDoElems (auxDo : Syntax) : List Syntax := getDoSeqElems (getDoSeq auxDo)
4593   let u ← mkTuple (← uvars.mapM mkIdentFromRef)
4594   match a, r, bc with
4595   | true, false, false =>
4596     if uvars.isEmpty then
4597       toDoElems (← `(do $term:term))
4598     else
4599       toDoElems (← `(do let r ← $term:term; $u:term := r.2; pure r.1))
4600   | false, true, false =>
4601     if uvars.isEmpty then
4602       toDoElems (← `(do let r ← $term:term; return r))
4603     else
4604       toDoElems (← `(do let r ← $term:term; $u:term := r.2; return r.1))
4605   | false, false, true => toDoElems <$>

```

```

4606   `(do let r ← $term:term;
4607       match r with
4608       | DoResultBC.«break» u => $u:term := u; break
4609       | DoResultBC.«continue» u => $u:term := u; continue)
4610 | true, true, false => toDoElems <$>
4611   `(do let r ← $term:term;
4612       match r with
4613       | DoResultPR.«pure» a u => $u:term := u; pure a
4614       | DoResultPR.«return» b u => $u:term := u; return b)
4615 | true, false, true => toDoElems <$>
4616   `(do let r ← $term:term;
4617       match r with
4618       | DoResultSBC.«pureReturn» a u => $u:term := u; pure a
4619       | DoResultSBC.«break» u => $u:term := u; break
4620       | DoResultSBC.«continue» u => $u:term := u; continue)
4621 | false, true, true => toDoElems <$>
4622   `(do let r ← $term:term;
4623       match r with
4624       | DoResultSBC.«pureReturn» a u => $u:term := u; return a
4625       | DoResultSBC.«break» u => $u:term := u; break
4626       | DoResultSBC.«continue» u => $u:term := u; continue)
4627 | true, true, true => toDoElems <$>
4628   `(do let r ← $term:term;
4629       match r with
4630       | DoResultPRBC.«pure» a u => $u:term := u; pure a
4631       | DoResultPRBC.«return» a u => $u:term := u; return a
4632       | DoResultPRBC.«break» u => $u:term := u; break
4633       | DoResultPRBC.«continue» u => $u:term := u; continue)
4634 | false, false, false => unreachable!
4635
4636 end ToTerm
4637
4638 def isMutableLet (doElem : Syntax) : Bool :=
4639   let kind := doElem.getKind
4640   (kind == `Lean.Parser.Term.doLetArrow || kind == `Lean.Parser.Term.doLet)
4641 &&
4642   !doElem[1].isNone
4643
4644 namespace ToCodeBlock
4645
4646 structure Context where
4647   ref          : Syntax
4648   m            : Syntax -- Syntax representing the monad associated with the do notation.
4649   mutableVars  : NameSet := {}
4650   insideFor    : Bool := false
4651
4652 abbrev M := ReaderT Context TermElabM

```

```

4653
4654 @[inline] def withNewMutableVars {α} (newVars : Array Name) (mutable : Bool) (x : M α) : M α :=
4655   withReader (fun ctx => if mutable then { ctx with mutableVars := insertVars ctx.mutableVars newVars } else ctx) x
4656
4657 def checkReassignable (xs : Array Name) : M Unit := do
4658   let throwInvalidReassignment (x : Name) : M Unit :=
4659     throwError "{x.simpMacroScopes}' cannot be reassigned"
4660   let ctx ← read
4661   for x in xs do
4662     unless ctx.mutableVars.contains x do
4663       throwInvalidReassignment x
4664
4665 @[inline] def withFor {α} (x : M α) : M α :=
4666   withReader (fun ctx => { ctx with insideFor := true }) x
4667
4668 structure ToForInTermResult where
4669   uvars      : Array Name
4670   term       : Syntax
4671
4672 def mkForInBody (x : Syntax) (forInBody : CodeBlock) : M ToForInTermResult := do
4673   let ctx ← read
4674   let uvars := forInBody.uvars
4675   let uvars := nameSetToArray uvars
4676   let term ← liftMacroM $ ToTerm.run forInBody.code ctx.m uvars (if hasReturn forInBody.code then ToTerm.Kind.forInWithReturn else ToTerm.Kind.pure)
4677   pure (uvars, term)
4678
4679 def ensureInsideFor : M Unit :=
4680   unless (← read).insideFor do
4681     throwError "invalid 'do' element, it must be inside 'for'"
4682
4683 def ensureEOS (doElems : List Syntax) : M Unit :=
4684   unless doElems.isEmpty do
4685     throwError "must be last element in a 'do' sequence"
4686
4687 private partial def expandLiftMethodAux (inQuot : Bool) (inBinder : Bool) : Syntax → StateT (List Syntax) MacroM Syntax
4688 | stx@(Syntax.node k args) =>
4689   if liftMethodDelimiter k then
4690     return stx
4691   else if k == `Lean.Parser.Term.liftMethod && !inQuot then withFreshMacroScope do
4692     if inBinder then
4693       Macro.throwErrorAt stx "cannot lift `(<- ...)` over a binder, this error usually happens when you are trying to lift a method name"
4694       let term := args[1]
4695       let term ← expandLiftMethodAux inQuot inBinder term
4696       let auxDoElem ← `(doElem| let a ← $term:term)
4697       modify fun s => s ++ [auxDoElem]
4698       `(a)
4699   else do

```

```

4700     let inAntiquot := stx.isAntiquot && !stx.isEscapedAntiquot
4701     let inBinder   := inBinder || (!inQuot && liftMethodForbiddenBinder k)
4702     let args ← args.mapM (expandLiftMethodAux (inQuot && !inAntiquot || stx.isQuot) inBinder)
4703     return Syntax.node k args
4704 | stx => pure stx
4705
4706 def expandLiftMethod (doElem : Syntax) : MacroM (List Syntax × Syntax) := do
4707   if !hasLiftMethod doElem then
4708     pure ([], doElem)
4709   else
4710     let (doElem, doElemsNew) ← (expandLiftMethodAux false false doElem).run []
4711     pure (doElemsNew, doElem)
4712
4713 def checkLetArrowRHS (doElem : Syntax) : M Unit := do
4714   let kind := doElem.getKind
4715   if kind == `Lean.Parser.Term.doLetArrow ||
4716     kind == `Lean.Parser.Term.doLet ||
4717     kind == `Lean.Parser.Term.doLetRec ||
4718     kind == `Lean.Parser.Term.doHave ||
4719     kind == `Lean.Parser.Term.doReassign ||
4720     kind == `Lean.Parser.Term.doReassignArrow then
4721     throwErrorAt doElem "invalid kind of value '{kind}' in an assignment"
4722
4723 /- Generate `CodeBlock` for `doReturn` which is of the form
4724 ```
4725 "return " >> optional termParser
4726 ```
4727 `doElems` is only used for sanity checking. -/
4728 def doReturnToCode (doReturn : Syntax) (doElems: List Syntax) : M CodeBlock := withRef doReturn do
4729   ensureEOS doElems
4730   let argOpt := doReturn[1]
4731   let arg ← if argOpt.isNone then liftMacroM mkUnit else pure argOpt[0]
4732   return mkReturn (← getRef) arg
4733
4734 structure Catch where
4735   x          : Syntax
4736   optType    : Syntax
4737   codeBlock  : CodeBlock
4738
4739 def getTryCatchUpdatedVars (tryCode : CodeBlock) (catches : Array Catch) (finallyCode? : Option CodeBlock) : NameSet :=
4740   let ws := tryCode.uvars
4741   let ws := catches.foldl (fun ws alt => union alt.codeBlock.uvars ws) ws
4742   let ws := match finallyCode? with
4743     | none   => ws
4744     | some c => union c.uvars ws
4745   ws
4746

```

```

4747 def tryCatchPred (tryCode : CodeBlock) (catches : Array Catch) (finallyCode? : Option CodeBlock) (p : Code → Bool) : Bool :=
4748   p tryCode.code ||
4749   catches.any (fun «catch» => p «catch».codeBlock.code) ||
4750   match finallyCode? with
4751   | none => false
4752   | some finallyCode => p finallyCode.code
4753
4754 mutual
4755   /- "Concatenate" `c` with `doSeqToCode doElems` -/
4756   partial def concatWith (c : CodeBlock) (doElems : List Syntax) : M CodeBlock :=
4757     match doElems with
4758     | [] => pure c
4759     | nextDoElem :: _ => do
4760       let k ← doSeqToCode doElems
4761       let ref := nextDoElem
4762       concat c ref none k
4763
4764   /- Generate `CodeBlock` for `doLetArrow; doElems`
4765       `doLetArrow` is of the form
4766       ```
4767       "let " >> optional "mut " >> (doIdDecl <|> doPatDecl)
4768       ```
4769       where
4770       ```
4771       def doIdDecl   := leading_parser ident >> optType >> leftArrow >> doElemParser
4772       def doPatDecl  := leading_parser termParser >> leftArrow >> doElemParser >> optional (" | " >> doElemParser)
4773       ```
4774   -/
4775   partial def doLetArrowToCode (doLetArrow : Syntax) (doElems : List Syntax) : M CodeBlock := do
4776     let ref      := doLetArrow
4777     let decl     := doLetArrow[2]
4778     if decl.getKind == `Lean.Parser.Term.doIdDecl then
4779       let y := decl[0].getId
4780       let doElem := decl[3]
4781       let k ← withNewMutableVars #[y] (isMutableLet doLetArrow) (doSeqToCode doElems)
4782       match isDoExpr? doElem with
4783       | some action => pure $ mkVarDeclCore #[y] doLetArrow k
4784       | none =>
4785         checkLetArrowRHS doElem
4786         let c ← doSeqToCode [doElem]
4787         match doElems with
4788         | []      => pure c
4789         | kRef::_ => concat c kRef y k
4790     else if decl.getKind == `Lean.Parser.Term.doPatDecl then
4791       let pattern := decl[0]
4792       let doElem  := decl[2]
4793       let optElse := decl[3]

```

```

4794   if optElse.isNone then withFreshMacroScope do
4795     let auxDo ←
4796       if isMutableLet doLetArrow then
4797         `(do let discr ← $doElem; let mut $pattern:term := discr)
4798       else
4799         `(do let discr ← $doElem; let $pattern:term := discr)
4800     doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
4801   else
4802     if isMutableLet doLetArrow then
4803       throwError "'mut' is currently not supported in let-decls with 'else' case"
4804     let contSeq := mkDoSeq doElems.toArray
4805     let elseSeq := mkSingletonDoSeq optElse[1]
4806     let auxDo ← `(do let discr ← $doElem; match discr with | $pattern:term => $contSeq | _ => $elseSeq)
4807     doSeqToCode <| getDoSeqElems (getDoSeq auxDo)
4808   else
4809     throwError "unexpected kind of 'do' declaration"
4810
4811
4812   /- Generate `CodeBlock` for `doReassignArrow; doElems`
4813   `doReassignArrow` is of the form
4814   ```
4815   (doIdDecl <|> doPatDecl)
4816   ```
4817   -/
4818   partial def doReassignArrowToCode (doReassignArrow : Syntax) (doElems : List Syntax) : M CodeBlock := do
4819     let ref := doReassignArrow
4820     let decl := doReassignArrow[0]
4821     if decl.getKind == `Lean.Parser.Term.doIdDecl then
4822       let doElem := decl[3]
4823       let y      := decl[0]
4824       let auxDo ← `(do let r ← $doElem; $y:ident := r)
4825       doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
4826     else if decl.getKind == `Lean.Parser.Term.doPatDecl then
4827       let pattern := decl[0]
4828       let doElem  := decl[2]
4829       let optElse := decl[3]
4830       if optElse.isNone then withFreshMacroScope do
4831         let auxDo ← `(do let discr ← $doElem; $pattern:term := discr)
4832         doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
4833       else
4834         throwError "reassignment with `|` (i.e., \"else clause\") is not currently supported"
4835     else
4836       throwError "unexpected kind of 'do' reassignment"
4837
4838   /- Generate `CodeBlock` for `doIf; doElems`
4839   `doIf` is of the form
4840   ```

```

```

4841 "if " >> optIdent >> termParser >> " then " >> doSeq
4842   >> many (group (try (group (" else " >> " if "))) >> optIdent >> termParser >> " then " >> doSeq))
4843   >> optional (" else " >> doSeq)
4844   `` -/
4845 partial def doIfToCode (doIf : Syntax) (doElems : List Syntax) : M CodeBlock := do
4846   let view ← liftMacroM $ mkDoIfView doIf
4847   let thenBranch ← doSeqToCode (getDoSeqElems view.thenBranch)
4848   let elseBranch ← doSeqToCode (getDoSeqElems view.elseBranch)
4849   let ite ← mkIte view.ref view.optIdent view.cond thenBranch elseBranch
4850   concatWith ite doElems
4851
4852 /- Generate `CodeBlock` for `doUnless; doElems`
4853 `doUnless` is of the form
4854 ``
4855 "unless " >> termParser >> "do " >> doSeq
4856 `` -/
4857 partial def doUnlessToCode (doUnless : Syntax) (doElems : List Syntax) : M CodeBlock := withRef doUnless do
4858   let ref := doUnless
4859   let cond := doUnless[1]
4860   let doSeq := doUnless[3]
4861   let body ← doSeqToCode (getDoSeqElems doSeq)
4862   let unlessCode ← liftMacroM <| mkUnless cond body
4863   concatWith unlessCode doElems
4864
4865 /- Generate `CodeBlock` for `doFor; doElems`
4866 `doFor` is of the form
4867 ``
4868 def doForDecl := leading_parser termParser >> " in " >> withForbidden "do" termParser
4869 def doFor := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq
4870 ``
4871 -/
4872 partial def doForToCode (doFor : Syntax) (doElems : List Syntax) : M CodeBlock := do
4873   let doForDecls := doFor[1].getSepArgs
4874   if doForDecls.size > 1 then
4875     /-
4876     Expand
4877     ``
4878     for x in xs, y in ys do
4879       body
4880     ``
4881     into
4882     ``
4883     let s := toStream ys
4884     for x in xs do
4885       match Stream.next? s with
4886       | none => break
4887       | some (y, s') =>

```



```

4888         s := s'
4889         body
4890     ...
4891 -/
4892 -- Extract second element
4893 let doForDecl := doForDecls[1]
4894 let y := doForDecl[0]
4895 let ys := doForDecl[2]
4896 let doForDecls := doForDecls.eraseIdx 1
4897 let body := doFor[3]
4898 withFreshMacroScope do
4899     let toStreamFn ← withRef ys `(toStream)
4900     let auxDo ←
4901         `(do let mut s := $toStreamFn.ident $ys
4902             for $doForDecls:doForDecl,* do
4903                 match Stream.next? s with
4904                 | none => break
4905                 | some ($y, s') =>
4906                     s := s'
4907                     do $body)
4908     doSeqToCode (getDoSeqElems (getDoSeq auxDo) ++ doElems)
4909 else withRef doFor do
4910     let x := doForDecls[0][0]
4911     let xs := doForDecls[0][2]
4912     let forElems := getDoSeqElems doFor[3]
4913     let forInBodyCodeBlock ← withFor (doSeqToCode forElems)
4914     let (uvars, forInBody) ← mkForInBody x forInBodyCodeBlock
4915     let uvarsTuple ← liftMacroM do mkTuple (← uvars.mapM mkIdentFromRef)
4916     if hasReturn forInBodyCodeBlock.code then
4917         let forInBody ← liftMacroM <| destructTuple uvars (← `(r)) forInBody
4918         let forInTerm ← `(forIn% $(xs) (MProd.mk none $uvarsTuple) fun $x r => let r := r.2; $forInBody)
4919         let auxDo ← `(do let r ← $forInTerm:term;
4920             $uvarsTuple:term := r.2;
4921             match r.1 with
4922             | none => Pure.pure (ensureExpectedType% "type mismatch, 'for'" PUnit.unit)
4923             | some a => return ensureExpectedType% "type mismatch, 'for'" a)
4924     doSeqToCode (getDoSeqElems (getDoSeq auxDo) ++ doElems)
4925 else
4926     let forInBody ← liftMacroM <| destructTuple uvars (← `(r)) forInBody
4927     let forInTerm ← `(forIn% $(xs) $uvarsTuple fun $x r => $forInBody)
4928     if doElems.isEmpty then
4929         let auxDo ← `(do let r ← $forInTerm:term;
4930             $uvarsTuple:term := r;
4931             Pure.pure (ensureExpectedType% "type mismatch, 'for'" PUnit.unit))
4932     doSeqToCode <| getDoSeqElems (getDoSeq auxDo)
4933 else
4934     let auxDo ← `(do let r ← $forInTerm:term; $uvarsTuple:term := r)

```

```

4935         doSeqToCode <| getDoSeqElems (getDoSeq auxDo) ++ doElems
4936
4937 /-- Generate `CodeBlock` for `doMatch; doElems` -/
4938 partial def doMatchToCode (doMatch : Syntax) (doElems: List Syntax) : M CodeBlock := do
4939   let ref      := doMatch
4940   let genParam := doMatch[1]
4941   let discrs    := doMatch[2]
4942   let optType   := doMatch[3]
4943   let matchAlts := doMatch[5][0].getArgs -- Array of `doMatchAlt`
4944   let alts ← matchAlts.mapM fun matchAlt => do
4945     let patterns := matchAlt[1]
4946     let vars ← getPatternsVarsEx patterns.getSepArgs
4947     let rhs := matchAlt[3]
4948     let rhs ← doSeqToCode (getDoSeqElems rhs)
4949     pure { ref := matchAlt, vars := vars, patterns := patterns, rhs := rhs : Alt CodeBlock }
4950   let matchCode ← mkMatch ref genParam discrs optType alts
4951   concatWith matchCode doElems
4952
4953 /--
4954   Generate `CodeBlock` for `doTry; doElems`
4955   ```
4956   def doTry := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally
4957   def doCatch      := leading_parser "catch " >> binderIdent >> optional (":" >> termParser) >> darrow >> doSeq
4958   def doCatchMatch := leading_parser "catch " >> doMatchAlts
4959   def doFinally    := leading_parser "finally " >> doSeq
4960   ```
4961 -/
4962 partial def doTryToCode (doTry : Syntax) (doElems: List Syntax) : M CodeBlock := do
4963   let ref := doTry
4964   let tryCode ← doSeqToCode (getDoSeqElems doTry[1])
4965   let optFinally := doTry[3]
4966   let catches ← doTry[2].getArgs.mapM fun catchStx => do
4967     if catchStx.getKind == `Lean.Parser.Term.doCatch then
4968       let x      := catchStx[1]
4969       let optType := catchStx[2]
4970       let c ← doSeqToCode (getDoSeqElems catchStx[4])
4971       pure { x := x, optType := optType, codeBlock := c : Catch }
4972     else if catchStx.getKind == `Lean.Parser.Term.doCatchMatch then
4973       let matchAlts := catchStx[1]
4974       let x ← `(ex)
4975       let auxDo ← `(do match ex with $matchAlts)
4976       let c ← doSeqToCode (getDoSeqElems (getDoSeq auxDo))
4977       pure { x := x, codeBlock := c, optType := mkNullNode : Catch }
4978     else
4979       throwError "unexpected kind of 'catch'"
4980   let finallyCode? ← if optFinally.isNone then pure none else some <$> doSeqToCode (getDoSeqElems optFinally[0][1])
4981   if catches.isEmpty && finallyCode?.isNone then

```

```

4982     throwError "invalid 'try', it must have a 'catch' or 'finally'"
4983   let ctx ← read
4984   let ws    := getTryCatchUpdatedVars tryCode catches finallyCode?
4985   let uvars := nameSetToArray ws
4986   let a     := tryCatchPred tryCode catches finallyCode? hasTerminalAction
4987   let r     := tryCatchPred tryCode catches finallyCode? hasReturn
4988   let bc    := tryCatchPred tryCode catches finallyCode? hasBreakContinue
4989   let toTerm (codeBlock : CodeBlock) : M Syntax := do
4990     let codeBlock ← liftM $ extendUpdatedVars codeBlock ws
4991     liftMacroM $ ToTerm.mkNestedTerm codeBlock.code ctx.m uvars a r bc
4992   let term ← toTerm tryCode
4993   let term ← catches.foldlM
4994     (fun term «catch» => do
4995       let catchTerm ← toTerm «catch».codeBlock
4996       if catch.optType.isNone then
4997         `(MonadExcept.tryCatch $term (fun $(«catch».x):ident => $catchTerm))
4998       else
4999         let type := «catch».optType[1]
5000         `(tryCatchThe $type $term (fun $(«catch».x):ident => $catchTerm)))
5001   term
5002   let term ← match finallyCode? with
5003   | none      => pure term
5004   | some finallyCode => withRef optFinally do
5005     unless finallyCode.uvars.isEmpty do
5006       throwError "'finally' currently does not support reassignments"
5007     if hasBreakContinueReturn finallyCode.code then
5008       throwError "'finally' currently does 'return', 'break', nor 'continue'"
5009     let finallyTerm ← liftMacroM <| ToTerm.run finallyCode.code ctx.m {} ToTerm.Kind.regular
5010     `(tryFinally $term $finallyTerm)
5011   let doElemsNew ← liftMacroM <| ToTerm.matchNestedTermResult term uvars a r bc
5012   doSeqToCode (doElemsNew ++ doElems)
5013
5014   partial def doSeqToCode : List Syntax → M CodeBlock
5015   | [] => do liftMacroM mkPureUnitAction
5016   | doElem::doElems => withIncRecDepth <| withRef doElem do
5017     checkMaxHeartbeats "'do'-expander"
5018     match (← liftMacroM <| expandMacro? doElem) with
5019     | some doElem => doSeqToCode (doElem::doElems)
5020     | none =>
5021     match (← liftMacroM <| expandDoIf? doElem) with
5022     | some doElem => doSeqToCode (doElem::doElems)
5023     | none =>
5024       let (liftedDoElems, doElem) ← liftM (liftMacroM <| expandLiftMethod doElem : TermElabM _)
5025       if !liftedDoElems.isEmpty then
5026         doSeqToCode (liftedDoElems ++ [doElem] ++ doElems)
5027       else
5028         let ref := doElem

```

```

5029 let concatWithRest (c : CodeBlock) : M CodeBlock := concatWith c doElems
5030 let k := doElem.getKind
5031 if k == `Lean.Parser.Term.doLet then
5032   let vars ← getDoLetVars doElem
5033   mkVarDeclCore vars doElem <$> withNewMutableVars vars (isMutableLet doElem) (doSeqToCode doElems)
5034 else if k == `Lean.Parser.Term.doHave then
5035   let var := getDoHaveVar doElem
5036   mkVarDeclCore #[var] doElem <$> (doSeqToCode doElems)
5037 else if k == `Lean.Parser.Term.doLetRec then
5038   let vars ← getDoLetRecVars doElem
5039   mkVarDeclCore vars doElem <$> (doSeqToCode doElems)
5040 else if k == `Lean.Parser.Term.doReassign then
5041   let vars ← getDoReassignVars doElem
5042   checkReassignable vars
5043   let k ← doSeqToCode doElems
5044   mkReassignCore vars doElem k
5045 else if k == `Lean.Parser.Term.doLetArrow then
5046   doLetArrowToCode doElem doElems
5047 else if k == `Lean.Parser.Term.doReassignArrow then
5048   doReassignArrowToCode doElem doElems
5049 else if k == `Lean.Parser.Term.doIf then
5050   doIfToCode doElem doElems
5051 else if k == `Lean.Parser.Term.doUnless then
5052   doUnlessToCode doElem doElems
5053 else if k == `Lean.Parser.Term.doFor then withFreshMacroScope do
5054   doForToCode doElem doElems
5055 else if k == `Lean.Parser.Term.doMatch then
5056   doMatchToCode doElem doElems
5057 else if k == `Lean.Parser.Term.doTry then
5058   doTryToCode doElem doElems
5059 else if k == `Lean.Parser.Term.doBreak then
5060   ensureInsideFor
5061   ensureEOS doElems
5062   return mkBreak ref
5063 else if k == `Lean.Parser.Term.doContinue then
5064   ensureInsideFor
5065   ensureEOS doElems
5066   return mkContinue ref
5067 else if k == `Lean.Parser.Term.doReturn then
5068   doReturnToCode doElem doElems
5069 else if k == `Lean.Parser.Term.doDbgTrace then
5070   return mkSeq doElem (← doSeqToCode doElems)
5071 else if k == `Lean.Parser.Term.doAssert then
5072   return mkSeq doElem (← doSeqToCode doElems)
5073 else if k == `Lean.Parser.Term.doNested then
5074   let nestedDoSeq := doElem[1]
5075   doSeqToCode (getDoSeqElems nestedDoSeq ++ doElems)

```

```

5076     else if k == `Lean.Parser.Term.doExpr then
5077         let term := doElem[0]
5078         if doElems.isEmpty then
5079             return mkTerminalAction term
5080         else
5081             return mkSeq term (← doSeqToCode doElems)
5082     else
5083         throwError "unexpected do-element\n{doElem}"
5084 end
5085
5086 def run (doStx : Syntax) (m : Syntax) : TermElabM CodeBlock :=
5087   (doSeqToCode <| getDoSeqElems <| getDoSeq doStx).run { ref := doStx, m := m }
5088
5089 end ToCodeBlock
5090
5091 /- Create a synthetic metavariable `?m` and assign `m` to it.
5092 We use `?m` to refer to `m` when expanding the `do` notation. -/
5093 private def mkMonadAlias (m : Expr) : TermElabM Syntax := do
5094   let result ← `( ?m)
5095   let mType ← inferType m
5096   let mvar ← elabTerm result mType
5097   assignExprMVar mvar.mvarId! m
5098   pure result
5099
5100 @[builtinTermElab «do»]
5101 def elabDo : TermElab := fun stx expectedType? => do
5102   tryPostponeIfNoneOrMVar expectedType?
5103   let bindInfo ← extractBind expectedType?
5104   let m ← mkMonadAlias bindInfo.m
5105   let codeBlock ← ToCodeBlock.run stx m
5106   let stxNew ← liftMacroM $ ToTerm.run codeBlock.code m
5107   trace[Elab.do] stxNew
5108   withMacroExpansion stx stxNew $ elabTermEnsuringType stxNew bindInfo.expectedType
5109
5110 end Do
5111
5112 builtin_initialize registerTraceClass `Elab.do
5113
5114 private def toDoElem (newKind : SyntaxNodeKind) : Macro := fun stx => do
5115   let stx := stx.setKind newKind
5116   withRef stx `(do $stx:doElem)
5117
5118 @[builtinMacro Lean.Parser.Term.termFor]
5119 def expandTermFor : Macro := toDoElem `Lean.Parser.Term.doFor
5120
5121 @[builtinMacro Lean.Parser.Term.termTry]
5122 def expandTermTry : Macro := toDoElem `Lean.Parser.Term.doTry

```

```

5123
5124 @[builtinMacro Lean.Parser.Term.termUnless]
5125 def expandTermUnless : Macro := toDoElem `Lean.Parser.Term.doUnless
5126
5127 @[builtinMacro Lean.Parser.Term.termReturn]
5128 def expandTermReturn : Macro := toDoElem `Lean.Parser.Term.doReturn
5129
5130 end Lean.Elab.Term
5131 ::::::::::::::
5132 Elab/Exception.lean
5133 ::::::::::::::
5134 /-
5135 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5136 Released under Apache 2.0 license as described in the file LICENSE.
5137 Authors: Leonardo de Moura
5138 -/
5139 import Lean.InternalExceptionId
5140 import Lean.Meta.Basic
5141
5142 namespace Lean.Elab
5143
5144 builtin_initialize postponeExceptionId : InternalExceptionId ← registerInternalExceptionId `postpone
5145 builtin_initialize unsupportedSyntaxExceptionId : InternalExceptionId ← registerInternalExceptionId `unsupportedSyntax
5146 builtin_initialize abortCommandExceptionId : InternalExceptionId ← registerInternalExceptionId `abortCommandElab
5147 builtin_initialize abortTermExceptionId : InternalExceptionId ← registerInternalExceptionId `abortTermElab
5148 builtin_initialize autoBoundImplicitExceptionId : InternalExceptionId ← registerInternalExceptionId `autoBoundImplicit
5149
5150 def throwPostpone [MonadExceptOf Exception m] : m  $\alpha$  :=
5151   throw $ Exception.internal postponeExceptionId
5152
5153 def throwUnsupportedSyntax [MonadExceptOf Exception m] : m  $\alpha$  :=
5154   throw $ Exception.internal unsupportedSyntaxExceptionId
5155
5156 def throwIllFormedSyntax [Monad m] [MonadError m] : m  $\alpha$  :=
5157   throwError "ill-formed syntax"
5158
5159 def throwAutoBoundImplicitLocal [MonadExceptOf Exception m] (n : Name) : m  $\alpha$  :=
5160   throw $ Exception.internal autoBoundImplicitExceptionId <| KVMMap.empty.insert `localId n
5161
5162 def isAutoBoundImplicitLocalException? (ex : Exception) : Option Name :=
5163   match ex with
5164   | Exception.internal id k =>
5165     if id == autoBoundImplicitExceptionId then
5166       some <| k.getName `localId `x
5167     else
5168       none
5169   | _ => none

```

```

5170
5171 def throwAlreadyDeclaredUniverseLevel [Monad m] [MonadError m] (u : Name) : m  $\alpha$  :=
5172   throwError "a universe level named '{u}' has already been declared"
5173
5174 -- Throw exception to abort elaboration of the current command without producing any error message
5175 def throwAbortCommand { $\alpha$  m} [MonadExcept Exception m] : m  $\alpha$  :=
5176   throw <| Exception.internal abortCommandExceptionId
5177
5178 -- Throw exception to abort elaboration of the current term without producing any error message
5179 def throwAbortTerm { $\alpha$  m} [MonadExcept Exception m] : m  $\alpha$  :=
5180   throw <| Exception.internal abortTermExceptionId
5181
5182 def isAbortExceptionId (id : InternalExceptionId) : Bool :=
5183   id == abortCommandExceptionId || id == abortTermExceptionId
5184
5185 def mkMessageCore (fileName : String) (fileMap : FileMap) (msgData : MessageData) (severity : MessageSeverity) (pos : String.Pos) : Message :=
5186   let pos := fileMap.toPosition pos
5187   { fileName := fileName, pos := pos, data := msgData, severity := severity }
5188
5189 end Lean.Elab
5190 ::::::::::::::
5191 Elab/Extra.lean
5192 ::::::::::::::
5193 /-
5194 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
5195 Released under Apache 2.0 license as described in the file LICENSE.
5196 Authors: Leonardo de Moura
5197 -/
5198 import Lean.Elab.App
5199
5200 /-
5201 Auxiliary elaboration functions: AKA custom elaborators
5202 -/
5203
5204 namespace Lean.Elab.Term
5205 open Meta
5206
5207 @[builtinTermElab binrel] def elabBinRel : TermElab := fun stx expectedType? => do
5208   match (← resolveId? stx[1]) with
5209   | some f =>
5210     let s ← saveState
5211     let (lhs, rhs) ← withSynthesize (mayPostpone := true) do
5212       let mut lhs ← elabTerm stx[2] none
5213       let mut rhs ← elabTerm stx[3] none
5214     if lhs.isAppOfArity `OfNat.ofNat 3 then
5215       lhs ← ensureHasType (← inferType rhs) lhs
5216     else if rhs.isAppOfArity `OfNat.ofNat 3 then

```

```

5217     rhs ← ensureHasType (← inferType lhs) rhs
5218     return (lhs, rhs)
5219   let lhsType ← inferType lhs
5220   let rhsType ← inferType rhs
5221   let (lhs, rhs) ←
5222     try
5223       pure (lhs, ← withRef stx[3] do ensureHasType lhsType rhs)
5224     catch _ =>
5225       try
5226         pure (← withRef stx[2] do ensureHasType rhsType lhs, rhs)
5227       catch _ =>
5228         s.restore
5229         -- Use default approach
5230         let lhs ← elabTerm stx[2] none
5231         let rhs ← elabTerm stx[3] none
5232         let lhsType ← inferType lhs
5233         let rhsType ← inferType rhs
5234         pure (lhs, ← withRef stx[3] do ensureHasType lhsType rhs)
5235   elabAppArgs f #[] #[Arg.expr lhs, Arg.expr rhs] expectedType? (explicit := false) (ellipsis := false)
5236 | none => throwUnknownConstant stx[1].getId
5237
5238 @[builtinTermElab forInMacro] def elabForIn : TermElab := fun stx expectedType? => do
5239   match stx with
5240   | `(forIn% $col $init $body) =>
5241     match (← isLocalIdent? col) with
5242     | none => elabTerm (← `(let col := $col; forIn% col $init $body)) expectedType?
5243     | some colFVar =>
5244       tryPostponeIfNoneOrMVar expectedType?
5245       let m ← getMonad expectedType?
5246       let colType ← inferType colFVar
5247       let elemType ← mkFreshExprMVar (mkSort (mkLevelSucc (← mkFreshLevelMVar)))
5248       let forInInstance ←
5249         try
5250           mkAppM `ForIn #[m, colType, elemType]
5251         catch
5252           ex => tryPostpone; throwError "failed to construct 'ForIn' instance for collection{indentExpr colType}\nand monad{indentExpr
5253 match (← trySynthInstance forInInstance) with
5254 | LOption.some val =>
5255   let ref ← getRef
5256   let forInFn ← mkConst ``forIn
5257   let namedArgs : Array NamedArg := #[
5258     { ref := ref, name := `m, val := Arg.expr m},
5259     { ref := ref, name := `p, val := Arg.expr colType},
5260     { ref := ref, name := `α, val := Arg.expr elemType},
5261     { ref := ref, name := `self, val := Arg.expr forInInstance},
5262     { ref := ref, name := `inst, val := Arg.expr val} ]
5263   elabAppArgs forInFn #[] #[Arg.stx col, Arg.stx init, Arg.stx body] expectedType? (explicit := false) (ellipsis := false)

```



```

5264         | LOption.undef    => tryPostpone; throwFailure forInInstance
5265         | LOption.none      => throwFailure forInInstance
5266     | _ => throwUnsupportedSyntax
5267 where
5268     getMonad (expectedType? : Option Expr) : TermElabM Expr := do
5269         match expectedType? with
5270         | none => throwError "invalid 'forIn%' notation, expected type is not available"
5271         | some expectedType =>
5272             match (← isTypeApp? expectedType) with
5273             | some (m, _) => return m
5274             | none => throwError "invalid 'forIn%' notation, expected type is not of the form `M α`{indentExpr expectedType}"
5275     throwFailure (forInInstance : Expr) : TermElabM Expr :=
5276         throwError "failed to synthesize instance for 'forIn%' notation{indentExpr forInInstance}"
5277
5278 end Lean.Elab.Term
5279 :::::::::::::::
5280 Elab/Frontend.lean
5281 :::::::::::::::
5282 /-
5283 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5284 Released under Apache 2.0 license as described in the file LICENSE.
5285 Authors: Leonardo de Moura, Sebastian Ullrich
5286 -/
5287 import Lean.Elab.Import
5288 import Lean.Elab.Command
5289 import Lean.Util.Profile
5290
5291 namespace Lean.Elab.Frontend
5292
5293 structure State where
5294     commandState : Command.State
5295     parserState   : Parser.ModuleParserState
5296     cmdPos        : String.Pos
5297     commands      : Array Syntax := #[]
5298
5299 structure Context where
5300     inputCtx : Parser.InputContext
5301
5302 abbrev FrontendM := ReaderT Context $ StateRefT State IO
5303
5304 def setCommandState (commandState : Command.State) : FrontendM Unit :=
5305     modify fun s => { s with commandState := commandState }
5306
5307 @[inline] def runCommandElabM (x : Command.CommandElabM α) : FrontendM α := do
5308     let ctx ← read
5309     let s ← get
5310     let cmdCtx : Command.Context := { cmdPos := s.cmdPos, fileName := ctx.inputCtx.fileName, fileMap := ctx.inputCtx.fileMap }

```

```

5311 match ← liftM <| EI0.toIO' <| (x cmdCtx).run s.commandState with
5312 | Except.error e      => throw <| IO.Error.userError s!"unexpected internal error: {← e.toMessageData.toString}"
5313 | Except.ok (a, sNew) => setCommandState sNew; return a
5314
5315 def elabCommandAtFrontend (stx : Syntax) : FrontendM Unit := do
5316   runCommandElabM do
5317     let infoTreeEnabled := (← getInfoState).enabled
5318     if checkTraceOption (← getOptions) `Elab.info then
5319       enableInfoTree
5320     Command.elabCommand stx
5321     enableInfoTree infoTreeEnabled
5322
5323 def updateCmdPos : FrontendM Unit := do
5324   modify fun s => { s with cmdPos := s.parserState.pos }
5325
5326 def getParserState : FrontendM Parser.ModuleParserState := do pure (← get).parserState
5327 def getCommandState : FrontendM Command.State := do pure (← get).commandState
5328 def setParserState (ps : Parser.ModuleParserState) : FrontendM Unit := modify fun s => { s with parserState := ps }
5329 def setMessages (msgs : MessageLog) : FrontendM Unit := modify fun s => { s with commandState := { s.commandState with messages := msgs }
5330 def getInputContext : FrontendM Parser.InputContext := do pure (← read).inputCtx
5331
5332 def processCommand : FrontendM Bool := do
5333   updateCmdPos
5334   let cmdState ← getCommandState
5335   let ictx ← getInputContext
5336   let pstate ← getParserState
5337   let scope := cmdState.scopes.head!
5338   let pmctx := { env := cmdState.env, options := scope.opts, currNamespace := scope.currNamespace, openDecls := scope.openDecls }
5339   let pos := ictx.fileMap.toPosition pstate.pos
5340   match profileit "parsing" scope.opts fun _ => Parser.parseCommand ictx pmctx pstate cmdState.messages with
5341   | (cmd, ps, messages) =>
5342     modify fun s => { s with commands := s.commands.push cmd }
5343     setParserState ps
5344     setMessages messages
5345     if Parser.isEOI cmd || Parser.isExitCommand cmd then
5346       pure true -- Done
5347     else
5348       profileitM IO.Error "elaboration" scope.opts <| elabCommandAtFrontend cmd
5349       pure false
5350
5351 partial def processCommands : FrontendM Unit := do
5352   let done ← processCommand
5353   unless done do
5354     processCommands
5355
5356 end Frontend
5357

```

```

5358 open Frontend
5359
5360 def IO.processCommands (inputCtx : Parser.InputContext) (parserState : Parser.ModuleParserState) (commandState : Command.State) : IO Sta
5361   let (_, s) ← (Frontend.processCommands.run { inputCtx := inputCtx }).run { commandState := commandState, parserState := parserState, c
5362   pure s
5363
5364 def process (input : String) (env : Environment) (opts : Options) (fileName : Option String := none) : IO (Environment × MessageLog) :=
5365   let fileName := fileName.getD "<input>"
5366   let inputCtx := Parser.mkInputContext input fileName
5367   let s ← IO.processCommands inputCtx { : Parser.ModuleParserState } (Command.mkState env {} opts)
5368   pure (s.commandState.env, s.commandState.messages)
5369
5370 builtin_initialize
5371   registerOption `printMessageEndPos { defValue := false, descr := "print end position of each message in addition to start position" }
5372   registerTraceClass `Elab.info
5373
5374 def getPrintMessageEndPos (opts : Options) : Bool :=
5375   opts.getBool `printMessageEndPos false
5376
5377 @[export lean_run_frontend]
5378 def runFrontend (input : String) (opts : Options) (fileName : String) (moduleName : Name) : IO (Environment × Bool) := do
5379   let inputCtx := Parser.mkInputContext input fileName
5380   let (header, parserState, messages) ← Parser.parseHeader inputCtx
5381   let (env, messages) ← processHeader header opts messages inputCtx
5382   let env := env.setMainModule moduleName
5383   let s ← IO.processCommands inputCtx parserState (Command.mkState env messages opts)
5384   for msg in s.commandState.messages.toList do
5385     IO.print (← msg.toString (includeEndPos := getPrintMessageEndPos opts))
5386   pure (s.commandState.env, !s.commandState.messages.hasErrors)
5387
5388 end Lean.Elab
5389 ::::::::::::::
5390 Elab/Import.lean
5391 ::::::::::::::
5392 /-
5393 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5394 Released under Apache 2.0 license as described in the file LICENSE.
5395 Authors: Leonardo de Moura, Sebastian Ullrich
5396 -/
5397 import Lean.Parser.Module
5398 namespace Lean.Elab
5399
5400 def headerToImports (header : Syntax) : List Import :=
5401   let imports := if header[0].isNone then [{ module := `Init : Import }] else []
5402   imports ++ header[1].getArgs.toList.map fun stx =>
5403     -- `stx` is of the form `(Module.import "import" "runtime"? id)
5404     let runtime := !stx[1].isNone

```

```

5405     let id      := stx[2].getId
5406     { module := id, runtimeOnly := runtime }
5407
5408 def processHeader (header : Syntax) (opts : Options) (messages : MessageLog) (inputCtx : Parser.InputContext) (trustLevel : UInt32 := 0)
5409   : IO (Environment × MessageLog) := do
5410   try
5411     let env ← importModules (headerToImports header) opts trustLevel
5412     pure (env, messages)
5413   catch e =>
5414     let env ← mkEmptyEnvironment
5415     let spos := header.getPos?.getD 0
5416     let pos  := inputCtx.fileMap.toPosition spos
5417     pure (env, messages.add { fileName := inputCtx.fileName, data := toString e, pos := pos })
5418
5419 def parseImports (input : String) (fileName : Option String := none) : IO (List Import × Position × MessageLog) := do
5420   let fileName := fileName.getD "<input>"
5421   let inputCtx := Parser.mkInputContext input fileName
5422   let (header, parserState, messages) ← Parser.parseHeader inputCtx
5423   pure (headerToImports header, inputCtx.fileMap.toPosition parserState.pos, messages)
5424
5425 @[export lean_print_imports]
5426 def printImports (input : String) (fileName : Option String) : IO Unit := do
5427   let (deps, pos, log) ← parseImports input fileName
5428   for dep in deps do
5429     let fname ← find0Lean dep.module
5430     IO.println fname
5431
5432 end Lean.Elab
5433 :::::::::::::::
5434 Elab/Inductive.lean
5435 :::::::::::::::
5436 /-
5437 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
5438 Released under Apache 2.0 license as described in the file LICENSE.
5439 Authors: Leonardo de Moura
5440 -/
5441 import Lean.Util.ReplaceLevel
5442 import Lean.Util.ReplaceExpr
5443 import Lean.Util.CollectLevelParams
5444 import Lean.Util.Constructions
5445 import Lean.Meta.CollectFVars
5446 import Lean.Meta.SizeOf
5447 import Lean.Elab.Command
5448 import Lean.Elab.DefView
5449 import Lean.Elab.DeclUtil
5450 import Lean.Elab.Deriving.Basic
5451

```

```

5452 namespace Lean.Elab.Command
5453 open Meta
5454
5455 builtin_initialize
5456   registerTraceClass `Elab.inductive
5457
5458 def checkValidInductiveModifier [Monad m] [MonadError m] (modifiers : Modifiers) : m Unit := do
5459   if modifiers.isNoncomputable then
5460     throwError "invalid use of 'noncomputable' in inductive declaration"
5461   if modifiers.isPartial then
5462     throwError "invalid use of 'partial' in inductive declaration"
5463   unless modifiers.attrs.size == 0 || (modifiers.attrs.size == 1 && modifiers.attrs[0].name == `class) do
5464     throwError "invalid use of attributes in inductive declaration"
5465
5466 def checkValidCtorModifier [Monad m] [MonadError m] (modifiers : Modifiers) : m Unit := do
5467   if modifiers.isNoncomputable then
5468     throwError "invalid use of 'noncomputable' in constructor declaration"
5469   if modifiers.isPartial then
5470     throwError "invalid use of 'partial' in constructor declaration"
5471   if modifiers.isUnsafe then
5472     throwError "invalid use of 'unsafe' in constructor declaration"
5473   if modifiers.attrs.size != 0 then
5474     throwError "invalid use of attributes in constructor declaration"
5475
5476 structure CtorView where
5477   ref      : Syntax
5478   modifiers : Modifiers
5479   inferMod  : Bool -- true if `{}` is used in the constructor declaration
5480   declName  : Name
5481   binders   : Syntax
5482   type?     : Option Syntax
5483   deriving  Inhabited
5484
5485 structure InductiveView where
5486   ref      : Syntax
5487   modifiers : Modifiers
5488   shortDeclName : Name
5489   declName   : Name
5490   levelNames : List Name
5491   binders    : Syntax
5492   type?      : Option Syntax
5493   ctors      : Array CtorView
5494   derivingClasses : Array DerivingClassView
5495   deriving    Inhabited
5496
5497 structure ElabHeaderResult where
5498   view      : InductiveView

```

```

5499 lctx      : LocalContext
5500 localInsts : LocalInstances
5501 params    : Array Expr
5502 type      : Expr
5503 deriving Inhabited
5504
5505 private partial def elabHeaderAux (views : Array InductiveView) (i : Nat) (acc : Array ElabHeaderResult) : TermElabM (Array ElabHeaderRe
5506   if h : i < views.size then
5507     let view := views.get (i, h)
5508     let acc ← Term.withAutoBoundImplicit <| Term.elabBinders view.binders.getArgs fun params => do
5509       match view.type? with
5510       | none =>
5511         let u ← mkFreshLevelMVar
5512         let type := mkSort u
5513         Term.synthesizeSyntheticMVarsNoPostponing
5514         let params ← Term.addAutoBoundImplicits params
5515         pure <| acc.push { lctx := (← getLCtx), localInsts := (← getLocalInstances), params := params, type := type, view := view }
5516       | some typeStx =>
5517         let type ← Term.elabType typeStx
5518         unless (← isTypeFormerType type) do
5519           throwErrorAt typeStx "invalid inductive type, resultant type is not a sort"
5520         Term.synthesizeSyntheticMVarsNoPostponing
5521         let params ← Term.addAutoBoundImplicits params
5522         pure <| acc.push { lctx := (← getLCtx), localInsts := (← getLocalInstances), params := params, type := type, view := view }
5523     elabHeaderAux views (i+1) acc
5524   else
5525     pure acc
5526
5527 private def checkNumParams (rs : Array ElabHeaderResult) : TermElabM Nat := do
5528   let numParams := rs[0].params.size
5529   for r in rs do
5530     unless r.params.size == numParams do
5531       throwErrorAt r.view.ref "invalid inductive type, number of parameters mismatch in mutually inductive datatypes"
5532   pure numParams
5533
5534 private def checkUnsafe (rs : Array ElabHeaderResult) : TermElabM Unit := do
5535   let isUnsafe := rs[0].view.modifiers.isUnsafe
5536   for r in rs do
5537     unless r.view.modifiers.isUnsafe == isUnsafe do
5538       throwErrorAt r.view.ref "invalid inductive type, cannot mix unsafe and safe declarations in a mutually inductive datatypes"
5539
5540 private def checkLevelNames (views : Array InductiveView) : TermElabM Unit := do
5541   if views.size > 1 then
5542     let levelNames := views[0].levelNames
5543     for view in views do
5544       unless view.levelNames == levelNames do
5545         throwErrorAt view.ref "invalid inductive type, universe parameters mismatch in mutually inductive datatypes"

```

```

5546
5547 private def mkTypeFor (r : ElabHeaderResult) : TermElabM Expr := do
5548   withLCtx r.lctx r.localInsts do
5549     mkForallFVars r.params r.type
5550
5551 private def throwUnexpectedInductiveType {α} : TermElabM α :=
5552   throwError "unexpected inductive resulting type"
5553
5554 private def eqvFirstTypeResult (firstType type : Expr) : MetaM Bool :=
5555   forallTelescopeReducing firstType fun _ firstTypeResult => isDefEq firstTypeResult type
5556
5557 -- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
5558 private partial def checkParamsAndResultType (type firstType : Expr) (numParams : Nat) : TermElabM Unit := do
5559   try
5560     forallTelescopeCompatible type firstType numParams fun _ type firstType =>
5561       forallTelescopeReducing type fun _ type =>
5562         forallTelescopeReducing firstType fun _ firstType => do
5563           match type with
5564           | Expr.sort .. =>
5565             unless (← isDefEq firstType type) do
5566               throwError "resulting universe mismatch, given{indentExpr type}\nexpected type{indentExpr firstType}"
5567           | _ =>
5568             throwError "unexpected inductive resulting type"
5569   catch
5570   | Exception.error ref msg => throw (Exception.error ref m!"invalid mutually inductive types, {msg}")
5571   | ex => throw ex
5572
5573 -- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
5574 private def checkHeader (r : ElabHeaderResult) (numParams : Nat) (firstType? : Option Expr) : TermElabM Expr := do
5575   let type ← mkTypeFor r
5576   match firstType? with
5577   | none => pure type
5578   | some firstType =>
5579     withRef r.view.ref $ checkParamsAndResultType type firstType numParams
5580     pure firstType
5581
5582 -- Auxiliary function for checking whether the types in mutually inductive declaration are compatible.
5583 private partial def checkHeaders (rs : Array ElabHeaderResult) (numParams : Nat) (i : Nat) (firstType? : Option Expr) : TermElabM Unit :=
5584   if i < rs.size then
5585     let type ← checkHeader rs[i] numParams firstType?
5586     checkHeaders rs numParams (i+1) type
5587
5588 private def elabHeader (views : Array InductiveView) : TermElabM (Array ElabHeaderResult) := do
5589   let rs ← elabHeaderAux views 0 #[]
5590   if rs.size > 1 then
5591     checkUnsafe rs
5592     let numParams ← checkNumParams rs

```

```

5593   checkHeaders rs numParams 0 none
5594   return rs
5595
5596 /- Create a local declaration for each inductive type in `rs`, and execute `x params indFVars`, where `params` are the inductive type pa
5597   `indFVars` are the new local declarations.
5598   We use the local context/instances and parameters of rs[0].
5599   Note that this method is executed after we executed `checkHeaders` and established all
5600   parameters are compatible. -/
5601 private partial def withInductiveLocalDecls {α} (rs : Array ElabHeaderResult) (x : Array Expr → Array Expr → TermElabM α) : TermElabM α
5602   let namesAndTypes ← rs.mapM r => do
5603     let type ← mkTypeFor r
5604     pure (r.view.shortDeclName, type)
5605   let r0 := rs[0]
5606   let params := r0.params
5607   withLCtx r0.lctx r0.localInsts $ withRef r0.view.ref do
5608     let rec loop (i : Nat) (indFVars : Array Expr) := do
5609       if h : i < namesAndTypes.size then
5610         let (id, type) := namesAndTypes.get (i, h)
5611         withLocalDeclD id type fun indFVar => loop (i+1) (indFVars.push indFVar)
5612       else
5613         x params indFVars
5614     loop 0 #[]
5615
5616 private def isInductiveFamily (numParams : Nat) (indFVar : Expr) : TermElabM Bool := do
5617   let indFVarType ← inferType indFVar
5618   forallTelescopeReducing indFVarType fun xs _ =>
5619     return xs.size > numParams
5620
5621 /-
5622   Elaborate constructor types.
5623
5624   Remark: we check whether the resulting type is correct, and the parameter occurrences are consistent, but
5625   we currently do not check for:
5626   - Positivity (it is a rare failure, and the kernel already checks for it).
5627   - Universe constraints (the kernel checks for it).
5628 -/
5629 private def elabCtors (indFVars : Array Expr) (indFVar : Expr) (params : Array Expr) (r : ElabHeaderResult) : TermElabM (List Constructor)
5630   let indFamily ← isInductiveFamily params.size indFVar
5631   r.view.ctors.toList.mapM fun ctorView =>
5632     Term.withAutoBoundImplicit <| Term.elabBinders ctorView.binders.getArgs fun ctorParams =>
5633       withRef ctorView.ref do
5634         let rec elabCtorType (k : Expr → TermElabM Constructor) : TermElabM Constructor := do
5635           match ctorView.type? with
5636           | none =>
5637             if indFamily then
5638               throwError "constructor resulting type must be specified in inductive family declaration"
5639             k <| mkAppN indFVar params

```



```

5640 | some ctorType =>
5641   let type ← Term.elabType ctorType
5642   Term.synthesizeSyntheticMVars (mayPostpone := true)
5643   let type ← instantiateMVars type
5644   let type ← checkParamOccs type
5645   forallTelescopeReducing type fun _ resultingType => do
5646     unless resultingType.getAppFn == indFVar do
5647       throwError "unexpected constructor resulting type{indentExpr resultingType}"
5648     unless (← isType resultingType) do
5649       throwError "unexpected constructor resulting type, type expected{indentExpr resultingType}"
5650   k type
5651 elabCtorType fun type => do
5652   Term.synthesizeSyntheticMVarsNoPostponing
5653   let ctorParams ← Term.addAutoBoundImplicits ctorParams
5654   let type ← mkForallFVars ctorParams type
5655   let type ← mkForallFVars params type
5656   return { name := ctorView.declName, type := type }
5657 where
5658 checkParamOccs (ctorType : Expr) : MetaM Expr :=
5659   let visit (e : Expr) : MetaM TransformStep := do
5660     let f := e.getAppFn
5661     if indFVars.contains f then
5662       let mut args := e.getAppArgs
5663       unless args.size ≥ params.size do
5664         throwError "unexpected inductive type occurrence{indentExpr e}"
5665       for i in [:params.size] do
5666         let param := params[i]
5667         let arg := args[i]
5668         unless (← isDefEq param arg) do
5669           throwError "inductive datatype parameter mismatch{indentExpr arg}\nexpected{indentExpr param}"
5670         args := args.set! i param
5671       return TransformStep.done (mkAppN f args)
5672     else
5673       return TransformStep.visit e
5674   transform ctorType (pre := visit)
5675
5676 /- Convert universe metavariables occurring in the `indTypes` into new parameters.
5677 Remark: if the resulting inductive datatype has universe metavariables, we will fix it later using
5678 `inferResultingUniverse`. -/
5679 private def levelMVarToParamAux (indTypes : List InductiveType) : StateRefT Nat TermElabM (List InductiveType) :=
5680   indTypes.mapM fun indType => do
5681     let type ← Term.levelMVarToParam' indType.type
5682     let ctors ← indType.ctors.mapM fun ctor => do
5683       let ctorType ← Term.levelMVarToParam' ctor.type
5684       pure { ctor with type := ctorType }
5685     pure { indType with ctors := ctors, type := type }
5686

```

```

5687 private def levelMVarToParam (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
5688   (levelMVarToParamAux indTypes).run' 1
5689
5690 private def getResultingUniverse : List InductiveType → TermElabM Level
5691 | []          => throwError "unexpected empty inductive declaration"
5692 | indType :: _ => forallTelescopeReducing indType.type fun _ r => do
5693   match r with
5694   | Expr.sort u _ => pure u
5695   | _             => throwError "unexpected inductive type resulting type"
5696
5697 def tmpIndParam := mkLevelParam `tmp_ind_univ_param
5698
5699 /-
5700   Return true if `u` is of the form `?m + k`.
5701   Return false if `u` does not contain universe metavariables.
5702   Throw exception otherwise. -/
5703 def shouldInferResultUniverse (u : Level) : TermElabM Bool := do
5704   let u ← instantiateLevelMVars u
5705   if u.hasMVar then
5706     match u.getLevelOffset with
5707     | Level.mvar mvarId _ => do
5708       Term.assignLevelMVar mvarId tmpIndParam
5709       pure true
5710     | _ =>
5711       throwError "cannot infer resulting universe level of inductive datatype, given level contains metavariables {mkSort u}, provide un.
5712   else
5713     pure false
5714
5715 /-
5716   Auxiliary function for `updateResultingUniverse`
5717   `accLevelAtCtor u r rOffset us` add `u` components to `us` if they are not already there and it is different from the resulting univer.
5718   If `u` is a `max`, then its components are recursively processed.
5719   If `u` is a `succ` and `rOffset > 0`, we process the `u`'s child using `rOffset-1`.
5720
5721   This method is used to infer the resulting universe level of an inductive datatype. -/
5722 def accLevelAtCtor : Level → Level → Nat → Array Level → TermElabM (Array Level)
5723 | Level.max u v _, r, rOffset, us => do let us ← accLevelAtCtor u r rOffset us; accLevelAtCtor v r rOffset us
5724 | Level.imax u v _, r, rOffset, us => do let us ← accLevelAtCtor u r rOffset us; accLevelAtCtor v r rOffset us
5725 | Level.zero _, _, _, us => pure us
5726 | Level.succ u _, r, rOffset+1, us => accLevelAtCtor u r rOffset us
5727 | u, r, rOffset, us =>
5728   if rOffset == 0 && u == r then pure us
5729   else if r.occurs u then throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
5730   else if rOffset > 0 then throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
5731   else if us.contains u then pure us
5732   else pure (us.push u)
5733

```

```

5734 /- Auxiliary function for `updateResultingUniverse` -/
5735 private partial def collectUniversesFromCtorTypeAux (r : Level) (rOffset : Nat) : Nat → Expr → Array Level → TermElabM (Array Level)
5736 | 0, Expr.forallE n d b c, us => do
5737   let u ← getLevel d
5738   let u ← instantiateLevelMVars u
5739   let us ← accLevelAtCtor u r rOffset us
5740   withLocalDecl n c.binderInfo d fun x =>
5741     let e := b.instantiate1 x
5742     collectUniversesFromCtorTypeAux r rOffset 0 e us
5743 | i+1, Expr.forallE n d b c, us => do
5744   withLocalDecl n c.binderInfo d fun x =>
5745     let e := b.instantiate1 x
5746     collectUniversesFromCtorTypeAux r rOffset i e us
5747 | _, _, us => pure us
5748
5749 /- Auxiliary function for `updateResultingUniverse` -/
5750 private partial def collectUniversesFromCtorType
5751 (r : Level) (rOffset : Nat) (ctorType : Expr) (numParams : Nat) (us : Array Level) : TermElabM (Array Level) :=
5752 collectUniversesFromCtorTypeAux r rOffset numParams ctorType us
5753
5754 /- Auxiliary function for `updateResultingUniverse` -/
5755 private partial def collectUniverses (r : Level) (rOffset : Nat) (numParams : Nat) (indTypes : List InductiveType) : TermElabM (Array Level) := do
5756 let mut us := #[]
5757 for indType in indTypes do
5758   for ctor in indType.ctors do
5759     us ← collectUniversesFromCtorType r rOffset ctor.type numParams us
5760 return us
5761
5762 def mkResultUniverse (us : Array Level) (rOffset : Nat) : Level :=
5763 if us.isEmpty && rOffset == 0 then
5764   levelOne
5765 else
5766   let r := Level.mkNaryMax us.toList
5767   if rOffset == 0 && !r.isZero && !r.isNeverZero then
5768     (mkLevelMax r levelOne).normalize
5769   else
5770     r.normalize
5771
5772 private def updateResultingUniverse (numParams : Nat) (indTypes : List InductiveType) : TermElabM (List InductiveType) := do
5773 let r ← getResultingUniverse indTypes
5774 let rOffset : Nat := r.getOffset
5775 let r : Level := r.getLevelOffset
5776 unless r.isParam do
5777   throwError "failed to compute resulting universe level of inductive datatype, provide universe explicitly"
5778 let us ← collectUniverses r rOffset numParams indTypes
5779 trace[Elab.inductive] "updateResultingUniverse us: {us}, r: {r}, rOffset: {rOffset}"
5780 let rNew := mkResultUniverse us rOffset

```

```

5781 let updateLevel (e : Expr) : Expr := e.replaceLevel fun u => if u == tmpIndParam then some rNew else none
5782 return indTypes.map fun indType =>
5783   let type := updateLevel indType.type;
5784   let ctors := indType.ctors.map fun ctor => { ctor with type := updateLevel ctor.type };
5785   { indType with type := type, ctors := ctors }
5786
5787 register_builtin_option bootstrap.inductiveCheckResultingUniverse : Bool := {
5788   defValue := true,
5789   group    := "bootstrap",
5790   descr    := "by default the `inductive/structure` commands report an error if the resulting universe is not zero, but may be zero for
5791 }
5792
5793 def checkResultingUniverse (u : Level) : TermElabM Unit := do
5794   if bootstrap.inductiveCheckResultingUniverse.get (← getOptions) then
5795     let u ← instantiateLevelMVars u
5796     if !u.isZero && !u.isNeverZero then
5797       throwError "invalid universe polymorphic type, the resultant universe is not Prop (i.e., 0), but it may be Prop for some parameter
5798
5799 private def checkResultingUniverses (indTypes : List InductiveType) : TermElabM Unit := do
5800   checkResultingUniverse (← getResultingUniverse indTypes)
5801
5802 private def collectUsed (indTypes : List InductiveType) : StateRefT CollectFVars.State MetaM Unit := do
5803   indTypes.forM fun indType => do
5804     Term.collectUsedFVars indType.type
5805     indType.ctors.forM fun ctor =>
5806       Term.collectUsedFVars ctor.type
5807
5808 private def removeUnused (vars : Array Expr) (indTypes : List InductiveType) : TermElabM (LocalContext × LocalInstances × Array Expr) :=
5809   let (_, used) ← (collectUsed indTypes).run {}
5810   Term.removeUnused vars used
5811
5812 private def withUsed {α} (vars : Array Expr) (indTypes : List InductiveType) (k : Array Expr → TermElabM α) : TermElabM α := do
5813   let (lctx, localInsts, vars) ← removeUnused vars indTypes
5814   withLCtx lctx localInsts $ k vars
5815
5816 private def updateParams (vars : Array Expr) (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
5817   indTypes.mapM fun indType => do
5818     let type ← mkForallFVars vars indType.type
5819     let ctors ← indType.ctors.mapM fun ctor => do
5820       let ctorType ← mkForallFVars vars ctor.type
5821       pure { ctor with type := ctorType }
5822     pure { indType with type := type, ctors := ctors }
5823
5824 private def collectLevelParamsInInductive (indTypes : List InductiveType) : Array Name := do
5825   let mut usedParams : CollectLevelParams.State := {}
5826   for indType in indTypes do
5827     usedParams := collectLevelParams usedParams indType.type

```

```

5828   for ctor in indType.ctors do
5829     usedParams := collectLevelParams usedParams ctor.type
5830   return usedParams.params
5831
5832 private def mkIndFVar2Const (views : Array InductiveView) (indFVars : Array Expr) (levelNames : List Name) : ExprMap Expr := do
5833   let levelParams := levelNames.map mkLevelParam;
5834   let mut m : ExprMap Expr := {}
5835   for i in [:views.size] do
5836     let view := views[i]
5837     let indFVar := indFVars[i]
5838     m := m.insert indFVar (mkConst view.declName levelParams)
5839   return m
5840
5841 /- Remark: `numVars` <= `numParams`. `numVars` is the number of context `variables` used in the inductive declaration,
5842    and `numParams` is `numVars` + number of explicit parameters provided in the declaration. -/
5843 private def replaceIndFVarsWithConsts (views : Array InductiveView) (indFVars : Array Expr) (levelNames : List Name)
5844   (numVars : Nat) (numParams : Nat) (indTypes : List InductiveType) : TermElabM (List InductiveType) :=
5845   let indFVar2Const := mkIndFVar2Const views indFVars levelNames
5846   indTypes.mapM fun indType => do
5847     let ctors <- indType.ctors.mapM fun ctor => do
5848       let type <- forallBoundedTelescope ctor.type numParams fun params type => do
5849         let type := type.replace fun e =>
5850           if !e.isFVar then
5851             none
5852           else match indFVar2Const.find? e with
5853             | none => none
5854             | some c => mkAppN c (params.extract 0 numVars)
5855         mkForallFVars params type
5856       pure { ctor with type := type }
5857     pure { indType with ctors := ctors }
5858
5859 abbrev Ctor2InferMod := Std.HashMap Name Bool
5860
5861 private def mkCtor2InferMod (views : Array InductiveView) : Ctor2InferMod := do
5862   let mut m := {}
5863   for view in views do
5864     for ctorView in view.ctors do
5865       m := m.insert ctorView.declName ctorView.inferMod
5866   return m
5867
5868 private def applyInferMod (views : Array InductiveView) (numParams : Nat) (indTypes : List InductiveType) : List InductiveType :=
5869   let ctor2InferMod := mkCtor2InferMod views
5870   indTypes.map fun indType =>
5871     let ctors := indType.ctors.map fun ctor =>
5872       let inferMod := ctor2InferMod.find! ctor.name -- true if `{}` was used
5873       let ctorType := ctor.type.inferImplicit numParams !inferMod
5874       { ctor with type := ctorType }

```

```

5875     { indType with ctors := ctors }
5876
5877 private def mkAuxConstructions (views : Array InductiveView) : TermElabM Unit := do
5878   let env ← getEnv
5879   let hasEq    := env.contains ``Eq
5880   let hasHEq   := env.contains ``HEq
5881   let hasUnit  := env.contains ``PUnit
5882   let hasProd  := env.contains ``Prod
5883   for view in views do
5884     let n := view.declName
5885     mkRecOn n
5886     if hasUnit then mkCasesOn n
5887     if hasUnit && hasEq && hasHEq then mkNoConfusion n
5888     if hasUnit && hasProd then mkBelow n
5889     if hasUnit && hasProd then mkIBelow n
5890   for view in views do
5891     let n := view.declName;
5892     if hasUnit && hasProd then mkBRecOn n
5893     if hasUnit && hasProd then mkBInductionOn n
5894
5895 private def mkInductiveDecl (vars : Array Expr) (views : Array InductiveView) : TermElabM Unit := do
5896   let view0 := views[0]
5897   let scopeLevelNames ← Term.getLevelNames
5898   checkLevelNames views
5899   let allUserLevelNames := view0.levelNames
5900   let isUnsafe          := view0.modifiers.isUnsafe
5901   withRef view0.ref <| Term.withLevelNames allUserLevelNames do
5902     let rs ← elabHeader views
5903     withInductiveLocalDecls rs fun params indFVars => do
5904       let numExplicitParams := params.size
5905       let mut indTypes := #[]
5906       for i in [:views.size] do
5907         let indFVar := indFVars[i]
5908         let r       := rs[i]
5909         let type    ← mkForallFVars params r.type
5910         let ctors   ← elabCtors indFVars indFVar params r
5911         indTypes := indTypes.push { name := r.view.declName, type := type, ctors := ctors : InductiveType }
5912       let indTypes := indTypes.toList
5913       Term.synthesizeSyntheticMVarsNoPostponing
5914       let u ← getResultingUniverse indTypes
5915       let inferLevel ← shouldInferResultUniverse u
5916       withUsed vars indTypes fun vars => do
5917         let numVars := vars.size
5918         let numParams := numVars + numExplicitParams
5919         let indTypes ← updateParams vars indTypes
5920         let indTypes ← levelMVarToParam indTypes
5921         let indTypes ← if inferLevel then updateResultingUniverse numParams indTypes else checkResultingUniverses indTypes; pure indType

```

```

5922   let usedLevelNames := collectLevelParamsInInductive indTypes
5923   match sortDeclLevelParams scopeLevelNames allUserLevelNames usedLevelNames with
5924   | Except.error msg      => throwError msg
5925   | Except.ok levelParams => do
5926     let indTypes ← replaceIndFVarsWithConsts views indFVars levelParams numVars numParams indTypes
5927     let indTypes := applyInferMod views numParams indTypes
5928     let decl := Declaration.inductDecl levelParams numParams indTypes isUnsafe
5929     Term.ensureNoUnassignedMVars decl
5930     addDecl decl
5931     mkAuxConstructions views
5932     -- We need to invoke `applyAttributes` because `class` is implemented as an attribute.
5933     for view in views do
5934       Term.applyAttributesAt view.declName view.modifiers.attrs AttributeApplicationTime.afterTypeChecking
5935
5936 private def applyDerivingHandlers (views : Array InductiveView) : CommandElabM Unit := do
5937   let mut processed : NameSet := {}
5938   for view in views do
5939     for classView in view.derivingClasses do
5940       let className := classView.className
5941       unless processed.contains className do
5942         processed := processed.insert className
5943         let mut declNames := #[]
5944         for view in views do
5945           if view.derivingClasses.any fun classView => classView.className == className then
5946             declNames := declNames.push view.declName
5947         classView.applyHandlers declNames
5948
5949 def elabInductiveViews (views : Array InductiveView) : CommandElabM Unit := do
5950   let view0 := views[0]
5951   let ref := view0.ref
5952   runTermElabM view0.declName fun vars => withRef ref do
5953     mkInductiveDecl vars views
5954     mkSizeOfInstances view0.declName
5955     applyDerivingHandlers views
5956
5957 end Lean.Elab.Command
5958 ::::::::::::::
5959 Elab/InfoTree.lean
5960 ::::::::::::::
5961 /-
5962 Copyright (c) 2020 Wojciech Nawrocki. All rights reserved.
5963 Released under Apache 2.0 license as described in the file LICENSE.
5964
5965 Authors: Wojciech Nawrocki, Leonardo de Moura
5966 -/
5967 import Lean.Data.Position
5968 import Lean.Expr

```

```

5969 import Lean.Message
5970 import Lean.Data.Json
5971 import Lean.Meta.Basic
5972 import Lean.Meta.PPGoal
5973
5974 namespace Lean.Elab
5975
5976 open Std (PersistentArray PersistentArray.empty PersistentHashMap)
5977
5978 /- Context after executing `liftTermElabM`.
5979    Note that the term information collected during elaboration may contain metavariables, and their
5980    assignments are stored at `mctx`. -/
5981 structure ContextInfo where
5982   env          : Environment
5983   fileMap      : FileMap
5984   mctx         : MetavarContext := {}
5985   options      : Options       := {}
5986   currNamespace : Name         := Name.anonymous
5987   openDecls    : List OpenDecl := []
5988   deriving Inhabited
5989
5990 structure TermInfo where
5991   lctx : LocalContext -- The local context when the term was elaborated.
5992   expr : Expr
5993   stx  : Syntax
5994   deriving Inhabited
5995
5996 structure CommandInfo where
5997   stx : Syntax
5998   deriving Inhabited
5999
6000 inductive CompletionInfo where
6001   | dot (termInfo : TermInfo) (field? : Option Syntax) (expectedType? : Option Expr)
6002   | id (stx : Syntax) (id : Name) (danglingDot : Bool) (lctx : LocalContext) (expectedType? : Option Expr)
6003   | namespaceId (stx : Syntax)
6004   | option (stx : Syntax)
6005   | endSection (stx : Syntax) (scopeNames : List String)
6006   | tactic (stx : Syntax) (goals : List MVarId)
6007   -- TODO `import`
6008
6009 def CompletionInfo.stx : CompletionInfo → Syntax
6010 | dot i .. => i.stx
6011 | id stx .. => stx
6012 | namespaceId stx => stx
6013 | option stx => stx
6014 | endSection stx .. => stx
6015 | tactic stx .. => stx

```



```

6016
6017 structure FieldInfo where
6018   name : Name
6019   lctx : LocalContext
6020   val  : Expr
6021   stx  : Syntax
6022   deriving Inhabited
6023
6024 /- We store the list of goals before and after the execution of a tactic.
6025 We also store the metavariable context at each time since, we want to unassigned metavariables
6026 at tactic execution time to be displayed as `?m...`. -/
6027 structure TacticInfo where
6028   mctxBefore : MetavarContext
6029   goalsBefore : List MVarId
6030   stx         : Syntax
6031   mctxAfter  : MetavarContext
6032   goalsAfter : List MVarId
6033   deriving Inhabited
6034
6035 structure MacroExpansionInfo where
6036   lctx : LocalContext -- The local context when the macro was expanded.
6037   before : Syntax
6038   after : Syntax
6039   deriving Inhabited
6040
6041 inductive Info where
6042   | ofTacticInfo (i : TacticInfo)
6043   | ofTermInfo (i : TermInfo)
6044   | ofCommandInfo (i : CommandInfo)
6045   | ofMacroExpansionInfo (i : MacroExpansionInfo)
6046   | ofFieldInfo (i : FieldInfo)
6047   | ofCompletionInfo (i : CompletionInfo)
6048   deriving Inhabited
6049
6050 inductive InfoTree where
6051   | context (i : ContextInfo) (t : InfoTree) -- The context object is created by `liftTermElabM` at `Command.lean`
6052   | node (i : Info) (children : PersistentArray InfoTree) -- The children contains information for nested term elaboration and tactic ev.
6053   | ofJson (j : Json) -- For user data
6054   | hole (mvarId : MVarId) -- The elaborator creates holes (aka metavariables) for tactics and postponed terms
6055   deriving Inhabited
6056
6057 partial def InfoTree.findInfo? (p : Info → Bool) (t : InfoTree) : Option InfoTree :=
6058   match t with
6059   | context _ t => findInfo? p t
6060   | node i ts =>
6061     if p i then
6062       some t

```

```

6063     else
6064         ts.findSome? (findInfo? p)
6065     | _ => none
6066
6067 structure InfoState where
6068     enabled      : Bool := false
6069     assignment   : PersistentHashMap MVarId InfoTree := {} -- map from holeId to InfoTree
6070     trees        : PersistentArray InfoTree := {}
6071     deriving Inhabited
6072
6073 class MonadInfoTree (m : Type → Type) where
6074     getInfoState      : m InfoState
6075     modifyInfoState   : (InfoState → InfoState) → m Unit
6076
6077 export MonadInfoTree (getInfoState modifyInfoState)
6078
6079 instance [MonadLift m n] [MonadInfoTree m] : MonadInfoTree n where
6080     getInfoState      := liftM (getInfoState : m _)
6081     modifyInfoState f := liftM (modifyInfoState f : m _)
6082
6083 partial def InfoTree.substitute (tree : InfoTree) (assignment : PersistentHashMap MVarId InfoTree) : InfoTree :=
6084     match tree with
6085     | node i c => node i <| c.map (substitute · assignment)
6086     | context i t => context i (substitute t assignment)
6087     | ofJson j => ofJson j
6088     | hole id => match assignment.find? id with
6089         | none      => hole id
6090         | some tree => substitute tree assignment
6091
6092 def ContextInfo.runMetaM (info : ContextInfo) (lctx : LocalContext) (x : MetaM α) : IO α := do
6093     let x := x.run { lctx := lctx } { mctx := info.mctx }
6094     let ((a, _), _) ← x.toIO { options := info.options, currNamespace := info.currNamespace, openDecls := info.openDecls } { env := info.e
6095     return a
6096
6097 def ContextInfo.toPPContext (info : ContextInfo) (lctx : LocalContext) : PPContext :=
6098     { env := info.env, mctx := info.mctx, lctx := lctx,
6099       opts := info.options, currNamespace := info.currNamespace, openDecls := info.openDecls }
6100
6101 def ContextInfo.ppSyntax (info : ContextInfo) (lctx : LocalContext) (stx : Syntax) : IO Format := do
6102     ppTerm (info.toPPContext lctx) stx
6103
6104 private def formatStxRange (ctx : ContextInfo) (stx : Syntax) : Format := do
6105     let pos      := stx.getPos?.getD 0
6106     let endPos   := stx.getTailPos?.getD pos
6107     return f!"{fmtPos pos stx.getHeadInfo}-{fmtPos endPos stx.getTailInfo}"
6108 where fmtPos pos info :=
6109     let pos := format <| ctx.fileMap.toPosition pos

```

```

6110     match info with
6111     | SourceInfo.original .. => pos
6112     | _                     => f!"{pos}†"
6113
6114 def TermInfo.runMetaM (info : TermInfo) (ctx : ContextInfo) (x : MetaM  $\alpha$ ) : IO  $\alpha$  :=
6115   ctx.runMetaM info.lctx x
6116
6117 def TermInfo.format (ctx : ContextInfo) (info : TermInfo) : IO Format := do
6118   info.runMetaM ctx do
6119     return f!"{← Meta.ppExpr info.expr} : {← Meta.ppExpr (← Meta.inferType info.expr)} @ {formatStxRange ctx info.stx}"
6120
6121 def CompletionInfo.format (ctx : ContextInfo) (info : CompletionInfo) : IO Format :=
6122   match info with
6123   | CompletionInfo.dot i (expectedType? := expectedType?) .. => return f!"[.] {← i.format ctx} : {expectedType?}"
6124   | CompletionInfo.id stx _ _ lctx expectedType? => ctx.runMetaM lctx do return f!"[.] {stx} : {expectedType?} @ {formatStxRange ctx info.stx}"
6125   | _ => return f!"[.] {info.stx} @ {formatStxRange ctx info.stx}"
6126
6127 def CommandInfo.format (ctx : ContextInfo) (info : CommandInfo) : IO Format := do
6128   return f!"command @ {formatStxRange ctx info.stx}"
6129
6130 def FieldInfo.format (ctx : ContextInfo) (info : FieldInfo) : IO Format := do
6131   ctx.runMetaM info.lctx do
6132     return f!"{info.name} : {← Meta.ppExpr (← Meta.inferType info.val)} := {← Meta.ppExpr info.val} @ {formatStxRange ctx info.stx}"
6133
6134 def ContextInfo.ppGoals (ctx : ContextInfo) (goals : List MVarId) : IO Format :=
6135   if goals.isEmpty then
6136     return "no goals"
6137   else
6138     ctx.runMetaM {} (return Std.Format.prefixJoin "\n" (← goals.mapM Meta.ppGoal))
6139
6140 def TacticInfo.format (ctx : ContextInfo) (info : TacticInfo) : IO Format := do
6141   let ctxB := { ctx with mctx := info.mctxBefore }
6142   let ctxA := { ctx with mctx := info.mctxAfter }
6143   let goalsBefore ← ctxB.ppGoals info.goalsBefore
6144   let goalsAfter  ← ctxA.ppGoals info.goalsAfter
6145   return f!"Tactic @ {formatStxRange ctx info.stx}\nbefore {goalsBefore}\nafter {goalsAfter}"
6146
6147 def MacroExpansionInfo.format (ctx : ContextInfo) (info : MacroExpansionInfo) : IO Format := do
6148   let before ← ctx.ppSyntax info.lctx info.before
6149   let after  ← ctx.ppSyntax info.lctx info.after
6150   return f!"Macro expansion\n{before}\n====>\n{after}"
6151
6152 def Info.format (ctx : ContextInfo) : Info → IO Format
6153 | ofTacticInfo i      => i.format ctx
6154 | ofTermInfo i        => i.format ctx
6155 | ofCommandInfo i     => i.format ctx
6156 | ofMacroExpansionInfo i => i.format ctx

```

```

6157 | ofFieldInfo i      => i.format ctx
6158 | ofCompletionInfo i  => i.format ctx
6159
6160 /--
6161   Helper function for propagating the tactic metavariable context to its children nodes.
6162   We need this function because we preserve `TacticInfo` nodes during backtracking *and* their
6163   children. Moreover, we backtrack the metavariable context to undo metavariable assignments.
6164   `TacticInfo` nodes save the metavariable context before/after the tactic application, and
6165   can be pretty printed without any extra information. This is not the case for `TermInfo` nodes.
6166   Without this function, the formatting method would often fail when processing `TermInfo` nodes
6167   that are children of `TacticInfo` nodes that have been preserved during backtracking.
6168   Saving the metavariable context at `TermInfo` nodes is also not a good option because
6169   at `TermInfo` creation time, the metavariable context often miss information, e.g.,
6170   a TC problem has not been resolved, a postponed subterm has not been elaborated, etc.
6171
6172   See `Term.SavedState.restore`.
6173 -/
6174 def Info.updateContext? : Option ContextInfo → Info → Option ContextInfo
6175 | some ctx, ofTacticInfo i => some { ctx with mctx := i.mctxAfter }
6176 | ctx?, _ => ctx?
6177
6178 partial def InfoTree.format (tree : InfoTree) (ctx? : Option ContextInfo := none) : IO Format := do
6179   match tree with
6180   | ofJson j      => return toString j
6181   | hole id       => return toString id
6182   | context i t   => format t i
6183   | node i cs     => match ctx? with
6184   | none => return "<context-not-available>"
6185   | some ctx =>
6186     let fmt ← i.format ctx
6187     if cs.size == 0 then
6188       return fmt
6189     else
6190       let ctx? := i.updateContext? ctx?
6191       return f!"{fmt}{Std.Format.nestD <| Std.Format.prefixJoin "\n" (← cs.toList.mapM fun c => format c ctx?)}"
6192
6193 section
6194 variable [Monad m] [MonadInfoTree m]
6195
6196 @[inline] private def modifyInfoTrees (f : PersistentArray InfoTree → PersistentArray InfoTree) : m Unit :=
6197   modifyInfoState fun s => { s with trees := f s.trees }
6198
6199 private def getResetInfoTrees : m (PersistentArray InfoTree) := do
6200   let trees := (← getInfoState).trees
6201   modifyInfoTrees fun _ => {}
6202   return trees
6203

```

```

6204 def pushInfoTree (t : InfoTree) : m Unit := do
6205   if (← getInfoState).enabled then
6206     modifyInfoTrees fun ts => ts.push t
6207
6208 def pushInfoLeaf (t : Info) : m Unit := do
6209   if (← getInfoState).enabled then
6210     pushInfoTree <| InfoTree.node (children := {}) t
6211
6212 def addCompletionInfo (info : CompletionInfo) : m Unit := do
6213   pushInfoLeaf <| Info.ofCompletionInfo info
6214
6215 def resolveGlobalConstNoOverloadWithInfo [MonadResolveName m] [MonadEnv m] [MonadError m] (stx : Syntax) (id := stx.getId) : m Name := do
6216   let n ← resolveGlobalConstNoOverload id
6217   if (← getInfoState).enabled then
6218     pushInfoLeaf <| Info.ofTermInfo { lctx := LocalContext.empty, expr := (← mkConstWithLevelParams n), stx := stx }
6219   return n
6220
6221 def resolveGlobalConstWithInfos [MonadResolveName m] [MonadEnv m] [MonadError m] (stx : Syntax) (id := stx.getId) : m (List Name) := do
6222   let ns ← resolveGlobalConst id
6223   if (← getInfoState).enabled then
6224     for n in ns do
6225       pushInfoLeaf <| Info.ofTermInfo { lctx := LocalContext.empty, expr := (← mkConstWithLevelParams n), stx := stx }
6226   return ns
6227
6228 def mkInfoNode (info : Info) : m Unit := do
6229   if (← getInfoState).enabled then
6230     modifyInfoTrees fun ts => PersistentArray.empty.push <| InfoTree.node info ts
6231
6232 @[inline] def withInfoContext' [MonadFinally m] (x : m  $\alpha$ ) (mkInfo :  $\alpha \rightarrow m$  (Sum Info MVarId)) : m  $\alpha$  := do
6233   if (← getInfoState).enabled then
6234     let treesSaved ← getResetInfoTrees
6235     Prod.fst <$> MonadFinally.tryFinally' x fun a? => do
6236       match a? with
6237       | none => modifyInfoTrees fun _ => treesSaved
6238       | some a =>
6239         let info ← mkInfo a
6240         modifyInfoTrees fun trees =>
6241           match info with
6242           | Sum.inl info => treesSaved.push <| InfoTree.node info trees
6243           | Sum.inr mvaId => treesSaved.push <| InfoTree.hole mvaId
6244   else
6245     x
6246
6247 @[inline] def withInfoTreeContext [MonadFinally m] (x : m  $\alpha$ ) (mkInfoTree : PersistentArray InfoTree  $\rightarrow m$  InfoTree) : m  $\alpha$  := do
6248   if (← getInfoState).enabled then
6249     let treesSaved ← getResetInfoTrees
6250     Prod.fst <$> MonadFinally.tryFinally' x fun _ => do

```

```

6251     let st ← getInfoState
6252     let tree ← mkInfoTree st.trees
6253     modifyInfoTrees fun _ => treesSaved.push tree
6254 else
6255     x
6256
6257 @[inline] def withInfoContext [MonadFinally m] (x : m α) (mkInfo : m Info) : m α := do
6258     withInfoTreeContext x (fun trees => do return InfoTree.node (← mkInfo) trees)
6259
6260 def getInfoHoleIdAssignment? (mvarId : MVarId) : m (Option InfoTree) :=
6261     return (← getInfoState).assignment[mvarId]
6262
6263 def assignInfoHoleId (mvarId : MVarId) (infoTree : InfoTree) : m Unit := do
6264     assert! (← getInfoHoleIdAssignment? mvarId).isNone
6265     modifyInfoState fun s => { s with assignment := s.assignment.insert mvarId infoTree }
6266 end
6267
6268 def withMacroExpansionInfo [MonadFinally m] [Monad m] [MonadInfoTree m] [MonadLCtx m] (before after : Syntax) (x : m α) : m α :=
6269     let mkInfo : m Info := do
6270         return Info.ofMacroExpansionInfo {
6271             lctx := (← getLCtx)
6272             before := before
6273             after := after
6274         }
6275     withInfoContext x mkInfo
6276
6277 @[inline] def withInfoHole [MonadFinally m] [Monad m] [MonadInfoTree m] (mvarId : MVarId) (x : m α) : m α := do
6278     if (← getInfoState).enabled then
6279         let treesSaved ← getResetInfoTrees
6280         Prod.fst <$> MonadFinally.tryFinally' x fun a? => modifyInfoState fun s =>
6281             if s.trees.size > 0 then
6282                 { s with trees := treesSaved, assignment := s.assignment.insert mvarId s.trees[s.trees.size - 1] }
6283             else
6284                 { s with trees := treesSaved }
6285     else
6286         x
6287
6288 def enableInfoTree [MonadInfoTree m] (flag := true) : m Unit :=
6289     modifyInfoState fun s => { s with enabled := flag }
6290
6291 def getInfoTrees [MonadInfoTree m] [Monad m] : m (PersistentArray InfoTree) :=
6292     return (← getInfoState).trees
6293
6294 end Lean.Elab
6295 ::::::::::::::
6296 Elab/LetRec.lean
6297 ::::::::::::::

```

```

6298 /-
6299 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
6300 Released under Apache 2.0 license as described in the file LICENSE.
6301 Authors: Leonardo de Moura
6302 -/
6303 import Lean.Elab.Attributes
6304 import Lean.Elab.Binders
6305 import Lean.Elab.DeclModifiers
6306 import Lean.Elab.SyntheticMVars
6307 import Lean.Elab.DeclarationRange
6308
6309 namespace Lean.Elab.Term
6310 open Meta
6311
6312 structure LetRecDeclView where
6313   ref          : Syntax
6314   attrs        : Array Attribute
6315   shortDeclName : Name
6316   declName     : Name
6317   numParams    : Nat
6318   type         : Expr
6319   mvar         : Expr -- auxiliary metavariable used to lift the 'let rec'
6320   valStx       : Syntax
6321
6322 structure LetRecView where
6323   decls : Array LetRecDeclView
6324   body  : Syntax
6325
6326 /- group ("let " >> nonReservedSymbol "rec ") >> sepBy1 (group (optional «attributes» >> letDecl)) ", " >> "; " >> termParser -/
6327 private def mkLetRecDeclView (letRec : Syntax) : TermElabM LetRecView := do
6328   let decls ← letRec[1][0].getSepArgs.mapM fun (attrDeclStx : Syntax) => do
6329     let docStr? ← expandOptDocComment? attrDeclStx[0]
6330     let attrOptStx := attrDeclStx[1]
6331     let attrs ← if attrOptStx.isNone then pure #[] else elabDeclAttrs attrOptStx[0]
6332     let decl := attrDeclStx[2][0]
6333     if decl.isOfKind `Lean.Parser.Term.letPatDecl then
6334       throwErrorAt decl "patterns are not allowed in 'let rec' expressions"
6335     else if decl.isOfKind `Lean.Parser.Term.letIdDecl || decl.isOfKind `Lean.Parser.Term.letEqnsDecl then
6336       let shortDeclName := decl[0].getId
6337       let currDeclName? ← getDeclName?
6338       let declName := currDeclName?.getD Name.anonymous ++ shortDeclName
6339       checkNotAlreadyDeclared declName
6340       applyAttributesAt declName attrs AttributeApplicationTime.beforeElaboration
6341       addDocString' declName docStr?
6342       addAuxDeclarationRanges declName decl decl[0]
6343       let binders := decl[1].getArgs
6344       let typeStx := expandOptType decl decl[2]

```

```

6345   let (type, numParams) ← elabBinders binders fun xs => do
6346     let type ← elabType typeStx
6347     registerCustomErrorIfMVar type typeStx "failed to infer 'let rec' declaration type"
6348     let type ← mkForallFVars xs type
6349     pure (type, xs.size)
6350   let mvar ← mkFreshExprMVar type MetavarKind.syntheticOpaque
6351   let valStx ←
6352     if decl.isOfKind `Lean.Parser.Term.letIdDecl then
6353       pure decl[4]
6354     else
6355       liftMacroM $ expandMatchAltsIntoMatch decl decl[3]
6356   pure {
6357     ref      := decl,
6358     attrs    := attrs,
6359     shortDeclName := shortDeclName,
6360     declName  := declName,
6361     numParams := numParams,
6362     type      := type,
6363     mvar      := mvar,
6364     valStx    := valStx
6365     : LetRecDeclView }
6366   else
6367     throwUnsupportedSyntax
6368   pure {
6369     decls := decls,
6370     body  := letRec[3]
6371   }
6372
6373 private partial def withAuxLocalDecls {α} (views : Array LetRecDeclView) (k : Array Expr → TermElabM α) : TermElabM α :=
6374   let rec loop (i : Nat) (fvars : Array Expr) : TermElabM α :=
6375     if h : i < views.size then
6376       let view := views.get (i, h)
6377       withLocalDeclD view.shortDeclName view.type fun fvar => loop (i+1) (fvars.push fvar)
6378     else
6379       k fvars
6380   loop 0 #[]
6381
6382 private def elabLetRecDeclValues (view : LetRecView) : TermElabM (Array Expr) :=
6383   view.decls.mapM fun view => do
6384     forallBoundedTelescope view.type view.numParams fun xs type =>
6385       withDeclName view.declName do
6386         let value ← elabTermEnsuringType view.valStx type
6387         mkLambdaFVars xs value
6388
6389 private def registerLetRecsToLift (views : Array LetRecDeclView) (fvars : Array Expr) (values : Array Expr) : TermElabM Unit := do
6390   let letRecsToLiftCurr := (← get).letRecsToLift
6391   for view in views do

```



```

6392     if letRecsToLiftCurr.any fun toLift => toLift.declName == view.declName then
6393         withRef view.ref do
6394             throwError "{view.declName}' has already been declared"
6395     let lctx ← getLCtx
6396     let localInsts ← getLocalInstances
6397     let toLift := views.mapIdx fun i view => {
6398         ref           := view.ref,
6399         fvarId        := fvars[i].fvarId!,
6400         attrs         := view.attrs,
6401         shortDeclName := view.shortDeclName,
6402         declName      := view.declName,
6403         lctx          := lctx,
6404         localInstances := localInsts,
6405         type          := view.type,
6406         val           := values[i],
6407         mvarId        := view.mvar.mvarId!
6408         : LetRecToLift }
6409     modify fun s => { s with letRecsToLift := toLift.toList ++ s.letRecsToLift }
6410
6411 @[builtinTermElab «letrec»] def elabLetRec : TermElab := fun stx expectedType? => do
6412     let view ← mkLetRecDeclView stx
6413     withAuxLocalDecls view.decls fun fvars => do
6414         let values ← elabLetRecDeclValues view
6415         let body ← elabTermEnsuringType view.body expectedType?
6416         registerLetRecsToLift view.decls fvars values
6417         let mvars := view.decls.map (·.mvar)
6418         pure $ mkAppN (← mkLambdaFVars fvars body) mvars
6419
6420 end Lean.Elab.Term
6421 ::::::::::::::
6422 Elab/Level.lean
6423 ::::::::::::::
6424 /-
6425 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6426 Released under Apache 2.0 license as described in the file LICENSE.
6427 Authors: Leonardo de Moura
6428 -/
6429 import Lean.Meta.LevelDefEq
6430 import Lean.Elab.Exception
6431 import Lean.Elab.Log
6432 import Lean.Elab.AutoBound
6433
6434 namespace Lean.Elab.Level
6435
6436 structure Context where
6437     options      : Options
6438     ref          : Syntax

```

```

6439 autoBoundImplicit : Bool
6440
6441 structure State where
6442   ngen      : NameGenerator
6443   mctx      : MetavarContext
6444   levelNames : List Name
6445
6446 abbrev LevelElabM := ReaderT Context (EStateM Exception State)
6447
6448 instance : MonadOptions LevelElabM where
6449   getOptions := return (← read).options
6450
6451 instance : MonadRef LevelElabM where
6452   getRef      := return (← read).ref
6453   withRef ref x := withReader (fun ctx => { ctx with ref := ref }) x
6454
6455 instance : AddMessageContext LevelElabM where
6456   addMessageContext msg := pure msg
6457
6458 instance : MonadNameGenerator LevelElabM where
6459   getNGen := return (← get).ngen
6460
6461   setNGen ngen := modify fun s => { s with ngen := ngen }
6462
6463 def mkFreshLevelMVar : LevelElabM Level := do
6464   let mvarId ← mkFreshId
6465   modify fun s => { s with mctx := s.mctx.addLevelMVarDecl mvarId }
6466   return mkLevelMVar mvarId
6467
6468 register_builtin_option maxUniverseOffset : Nat := {
6469   defValue := 32
6470   descr   := "maximum universe level offset"
6471 }
6472
6473 private def checkUniverseOffset [Monad m] [MonadError m] [MonadOptions m] (n : Nat) : m Unit := do
6474   let max := maxUniverseOffset.get (← getOptions)
6475   unless n <= max do
6476     throwError "maximum universe level offset threshold ({max}) has been reached, you can increase the limit using option `set_option ma
6477
6478 partial def elabLevel (stx : Syntax) : LevelElabM Level := withRef stx do
6479   let kind := stx.getKind
6480   if kind == `Lean.Parser.Level.paren then
6481     elabLevel (stx.getArg 1)
6482   else if kind == `Lean.Parser.Level.max then
6483     let args := stx.getArg 1 |>.getArgs
6484     args[:args.size - 1].foldrM (init := ← elabLevel args.back) fun stx lvl =>
6485       return mkLevelMax' (← elabLevel stx) lvl

```

```

6486 else if kind == `Lean.Parser.Level.imax then
6487   let args := stx.getArg 1 |>.getArgs
6488   args[:args.size - 1].foldrM (init := ← elabLevel args.back) fun stx lvl =>
6489     return mkLevelIMax' (← elabLevel stx) lvl
6490 else if kind == `Lean.Parser.Level.hole then
6491   mkFreshLevelMVar
6492 else if kind == numLitKind then
6493   match stx.isNatLit? with
6494   | some val => checkUniverseOffset val; return Level.ofNat val
6495   | none     => throwIllFormedSyntax
6496 else if kind == identKind then
6497   let paramName := stx.getId
6498   unless (← get).levelNames.contains paramName do
6499     if (← read).autoBoundImplicit && isValidAutoBoundLevelName paramName then
6500       modify fun s => { s with levelNames := paramName :: s.levelNames }
6501     else
6502       throwError "unknown universe level '{paramName}'"
6503   return mkLevelParam paramName
6504 else if kind == `Lean.Parser.Level.addLit then
6505   let lvl ← elabLevel (stx.getArg 0)
6506   match stx.getArg 2 |>.isNatLit? with
6507   | some val => checkUniverseOffset val; return lvl.addOffset val
6508   | none     => throwIllFormedSyntax
6509 else
6510   throwError "unexpected universe level syntax kind"
6511
6512 end Lean.Elab.Level
6513 ::::::::::::::
6514 Elab/Log.lean
6515 ::::::::::::::
6516 /-
6517 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6518 Released under Apache 2.0 license as described in the file LICENSE.
6519 Authors: Leonardo de Moura
6520 -/
6521 import Lean.Elab.Util
6522 import Lean.Util.Sorry
6523 import Lean.Elab.Exception
6524
6525 namespace Lean.Elab
6526
6527 class MonadFileMap (m : Type → Type) where
6528   getFileMap : m FileMap
6529
6530 export MonadFileMap (getFileMap)
6531
6532 class MonadLog (m : Type → Type) extends MonadFileMap m where

```

```

6533 getRef      : m Syntax
6534 getFileName : m String
6535 logMessage  : Message → m Unit
6536
6537 export MonadLog (getFileName logMessage)
6538
6539 instance (m n) [MonadLift m n] [MonadLog m] : MonadLog n where
6540   getRef      := liftM (MonadLog.getRef : m _)
6541   getFileMap  := liftM (getFileMap : m _)
6542   getFileName := liftM (getFileName : m _)
6543   logMessage  := fun msg => liftM (logMessage msg : m _)
6544
6545 variable {m : Type → Type} [Monad m] [MonadLog m] [AddMessageContext m]
6546
6547 def getRefPos : m String.Pos := do
6548   let ref ← MonadLog.getRef
6549   return ref.getPos?.getD 0
6550
6551 def getRefPosition : m Position := do
6552   let fileMap ← getFileMap
6553   return fileMap.toPosition (← getRefPos)
6554
6555 def logAt (ref : Syntax) (msgData : MessageData) (severity : MessageSeverity := MessageSeverity.error) : m Unit :=
6556   unless severity == MessageSeverity.error && msgData.hasSyntheticSorry do
6557     let ref      := replaceRef ref (← MonadLog.getRef)
6558     let pos      := ref.getPos?.getD 0
6559     let endPos   := ref.getTailPos?.getD pos
6560     let fileMap  ← getFileMap
6561     let msgData  ← addMessageContext msgData
6562     logMessage { fileName := (← getFileName), pos := fileMap.toPosition pos, endPos := fileMap.toPosition endPos, data := msgData, sever
6563
6564 def logErrorAt (ref : Syntax) (msgData : MessageData) : m Unit :=
6565   logAt ref msgData MessageSeverity.error
6566
6567 def logWarningAt (ref : Syntax) (msgData : MessageData) : m Unit :=
6568   logAt ref msgData MessageSeverity.warning
6569
6570 def logInfoAt (ref : Syntax) (msgData : MessageData) : m Unit :=
6571   logAt ref msgData MessageSeverity.information
6572
6573 def log (msgData : MessageData) (severity : MessageSeverity := MessageSeverity.error) : m Unit := do
6574   let ref ← MonadLog.getRef
6575   logAt ref msgData severity
6576
6577 def logError (msgData : MessageData) : m Unit :=
6578   log msgData MessageSeverity.error
6579

```

```

6580 def logWarning (msgData : MessageData) : m Unit :=
6581   log msgData MessageSeverity.warning
6582
6583 def logInfo (msgData : MessageData) : m Unit :=
6584   log msgData MessageSeverity.information
6585
6586 def logException [MonadLiftT IO m] (ex : Exception) : m Unit := do
6587   match ex with
6588   | Exception.error ref msg => logErrorAt ref msg
6589   | Exception.internal id _ =>
6590     unless isAbortExceptionId id do
6591       let name ← id.getName
6592       logError m!"internal exception: {name}"
6593
6594 def logTrace (cls : Name) (msgData : MessageData) : m Unit := do
6595   logInfo (MessageData.tagged cls m!"[{cls}] {msgData}")
6596
6597 @[inline] def trace [MonadOptions m] (cls : Name) (msg : Unit → MessageData) : m Unit := do
6598   if checkTraceOption (← getOptions) cls then
6599     logTrace cls (msg ())
6600
6601 def logDbgTrace [MonadOptions m] (msg : MessageData) : m Unit := do
6602   trace `Elab.debug fun _ => msg
6603
6604 def logUnknownDecl (declName : Name) : m Unit :=
6605   logError m!"unknown declaration '{declName}'"
6606
6607 end Lean.Elab
6608 ::::::::::::::
6609 Elab/Match.lean
6610 ::::::::::::::
6611 /-
6612 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
6613 Released under Apache 2.0 license as described in the file LICENSE.
6614 Authors: Leonardo de Moura
6615 -/
6616 import Lean.Util.CollectFVars
6617 import Lean.Meta.Match.MatchPatternAttr
6618 import Lean.Meta.Match.Match
6619 import Lean.Meta.SortLocalDecls
6620 import Lean.Meta.GeneralizeVars
6621 import Lean.Elab.SyntheticMVars
6622 import Lean.Elab.App
6623 import Lean.Parser.Term
6624
6625 namespace Lean.Elab.Term
6626 open Meta

```

```

6627 open Lean.Parser.Term
6628
6629 /- This modules assumes "match"-expressions use the following syntax.
6630
6631 ```lean
6632 def matchDiscr := leading_parser optional (try (ident >> checkNoWsBefore "no space before ':'" >> ":")) >> termParser
6633
6634 def «match» := leading_parser:leadPrec "match " >> sepBy1 matchDiscr ", " >> optType >> " with " >> matchAlts
6635 ```
6636 -/
6637
6638 structure MatchAltView where
6639   ref      : Syntax
6640   patterns : Array Syntax
6641   rhs      : Syntax
6642   deriving Inhabited
6643
6644 private def expandSimpleMatch (stx discr lhsVar rhs : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
6645   let newStx ← `(let $lhsVar := $discr; $rhs)
6646   withMacroExpansion stx newStx <| elabTerm newStx expectedType?
6647
6648 private def elabDiscrsWitMatchType (discrStxs : Array Syntax) (matchType : Expr) (expectedType : Expr) : TermElabM (Array Expr × Bool) :=
6649   let mut discrs := #[]
6650   let mut i := 0
6651   let mut matchType := matchType
6652   let mut isDep := false
6653   for discrStx in discrStxs do
6654     i := i + 1
6655     matchType ← whnf matchType
6656     match matchType with
6657     | Expr.forallE _ d b _ =>
6658       let discr ← fullApproxDefEq <| elabTermEnsuringType discrStx[1] d
6659       trace[Elab.match] "discr #{i} {discr} : {d}"
6660       if b.hasLooseBVars then
6661         isDep := true
6662         matchType ← b.instantiate1 discr
6663         discrs := discrs.push discr
6664     | _ =>
6665       throwError "invalid type provided to match-expression, function type with arity #{discrStxs.size} expected"
6666   pure (discrs, isDep)
6667
6668 private def mkUserNameFor (e : Expr) : TermElabM Name := do
6669   match e with
6670   /- Remark: we use `mkFreshUserName` to make sure we don't add a variable to the local context that can be resolved to `e`. -/
6671   | Expr.fvar fvarId _ => mkFreshUserName ((← getLocalDecl fvarId).userName)
6672   | _                  => mkFreshBinderName
6673

```

```

6674 /- Return true iff `n` is an auxiliary variable created by `expandNonAtomicDiscrs?` -/
6675 def isAuxDiscrName (n : Name) : Bool :=
6676   n.hasMacroScopes && n.eraseMacroScopes == `_discr
6677
6678 /- We treat `@x` as atomic to avoid unnecessary extra local declarations from being
6679   inserted into the local context. Recall that `expandMatchAltsIntoMatch` uses `@` modifier.
6680   Thus this is kind of discriminant is quite common.
6681
6682   Remark: if the discriminat is `Syntax.missing`, we abort the elaboration of the `match`-expression.
6683   This can happen due to error recovery. Example
6684   ```
6685   example : (p v p) → p := fun h => match
6686     ```
6687   If we don't abort, the elaborator loops because we will keep trying to expand
6688   ```
6689   match
6690     ```
6691   into
6692     ```
6693   let d := <Syntax.missing>; match
6694     ```
6695   Recall that `Syntax.setArg stx i arg` is a no-op when `i` is out-of-bounds. -/
6696 def isAtomicDiscr? (discr : Syntax) : TermElabM (Option Expr) := do
6697   match discr with
6698   | `($x:ident) => isLocalIdent? x
6699   | `(@$x:ident) => isLocalIdent? x
6700   | _ => if discr.isMissing then throwAbortTerm else return none
6701
6702 -- See expandNonAtomicDiscrs?
6703 private def elabAtomicDiscr (discr : Syntax) : TermElabM Expr := do
6704   let term := discr[1]
6705   match (← isAtomicDiscr? term) with
6706   | some e@(Expr.fvar fvarId _) =>
6707     let localDecl ← getLocalDecl fvarId
6708     if !isAuxDiscrName localDecl.userName then
6709       pure e -- it is not an auxiliary local created by `expandNonAtomicDiscrs?`
6710     else
6711       pure localDecl.value
6712   | _ => throwErrorAt discr "unexpected discriminant"
6713
6714 structure ElabMatchTypeAndDiscsResult where
6715   discrs      : Array Expr
6716   matchType   : Expr
6717   /- `true` when performing dependent elimination. We use this to decide whether we optimize the "match unit" case.
6718   See `isMatchUnit?`. -/
6719   isDep       : Bool
6720   alts        : Array MatchAltView

```

```

6721
6722 private def elabMatchTypeAndDiscrs (discrStxs : Array Syntax) (matchOptType : Syntax) (matchAltViews : Array MatchAltView) (expectedType
6723   : TermElabM ElabMatchTypeAndDiscrsResult := do
6724   let numDiscrs := discrStxs.size
6725   if matchOptType.isNone then
6726     let rec loop (i : Nat) (discrs : Array Expr) (matchType : Expr) (isDep : Bool) (matchAltViews : Array MatchAltView) := do
6727       match i with
6728       | 0 => return { discrs := discrs.reverse, matchType := matchType, isDep := isDep, alts := matchAltViews }
6729       | i+1 =>
6730         let discrStx := discrStxs[i]
6731         let discr ← elabAtomicDiscr discrStx
6732         let discr ← instantiateMVars discr
6733         let discrType ← inferType discr
6734         let discrType ← instantiateMVars discrType
6735         let matchTypeBody ← kabstract matchType discr
6736         let isDep := isDep || matchTypeBody.hasLooseBVars
6737         let userName ← mkUserNameFor discr
6738         if discrStx[0].isNone then
6739           loop i (discrs.push discr) (Lean.mkForall userName BinderInfo.default discrType matchTypeBody) isDep matchAltViews
6740         else
6741           let identStx := discrStx[0][0]
6742           withLocalDeclD userName discrType fun x => do
6743             let eqType ← mkEq discr x
6744             withLocalDeclD identStx.getId eqType fun h => do
6745               let matchTypeBody := matchTypeBody.instantiate1 x
6746               let matchType ← mkForallFVars #[x, h] matchTypeBody
6747               let refl ← mkEqRefl discr
6748               let discrs := (discrs.push refl).push discr
6749               let matchAltViews := matchAltViews.map fun altView =>
6750                 { altView with patterns := altView.patterns.insertAt (i+1) identStx }
6751               loop i discrs matchType isDep matchAltViews
6752     loop discrStxs.size (discrs := #[]) (isDep := false) expectedType matchAltViews
6753   else
6754     let matchTypeStx := matchOptType[0][1]
6755     let matchType ← elabType matchTypeStx
6756     let (discrs, isDep) ← elabDiscrsWitMatchType discrStxs matchType expectedType
6757     return { discrs := discrs, matchType := matchType, isDep := isDep, alts := matchAltViews }
6758
6759 def expandMacrosInPatterns (matchAlts : Array MatchAltView) : MacroM (Array MatchAltView) := do
6760   matchAlts.mapM fun matchAlt => do
6761     let patterns ← matchAlt.patterns.mapM expandMacros
6762     pure { matchAlt with patterns := patterns }
6763
6764 private def getMatchGeneralizing? : Syntax → Option Bool
6765 | `(match (generalizing := true) $discrs,* $[: $ty?]? with $alts:matchAlt*) => some true
6766 | `(match (generalizing := false) $discrs,* $[: $ty?]? with $alts:matchAlt*) => some false
6767 | _ => none

```



```

6768
6769 /- Given `stx` a match-expression, return its alternatives. -/
6770 private def getMatchAlts : Syntax → Array MatchAltView
6771 | `(match $[$gen]? $discrs,* $[: $ty]? with $alts:matchAlt*) =>
6772   alts.filterMap fun alt => match alt with
6773   | `(matchAltExpr| | $patterns,* => $rhs) => some {
6774     ref      := alt,
6775     patterns := patterns,
6776     rhs      := rhs
6777   }
6778 | _ => none
6779 | _ => #[]
6780
6781 /--
6782   Auxiliary annotation used to mark terms marked with the "inaccessible" annotation `.(t)` and
6783   `_` in patterns. -/
6784 def mkInaccessible (e : Expr) : Expr :=
6785   mkAnnotation `_inaccessible e
6786
6787 def inaccessible? (e : Expr) : Option Expr :=
6788   annotation? `_inaccessible e
6789
6790 inductive PatternVar where
6791 | localVar (userName : Name)
6792 -- anonymous variables (`_`) are encoded using metavariables
6793 | anonymousVar (mvarId : MVarId)
6794
6795 instance : ToString PatternVar := {fun
6796 | PatternVar.localVar x      => toString x
6797 | PatternVar.anonymousVar mvarId => s!"?m{mvarId}"}
6798
6799 builtin_initialize Parser.registerBuiltinNodeKind `MVarWithIdKind
6800
6801 /--
6802   Create an auxiliary Syntax node wrapping a fresh metavariable id.
6803   We use this kind of Syntax for representing `_` occurring in patterns.
6804   The metavariables are created before we elaborate the patterns into `Expr`s. -/
6805 private def mkMVarSyntax : TermElabM Syntax := do
6806   let mvarId ← mkFreshId
6807   return Syntax.node `MVarWithIdKind #[Syntax.node mvarId #[]]
6808
6809 /-- Given a syntax node constructed using `mkMVarSyntax`, return its MVarId -/
6810 private def getMVarSyntaxMVarId (stx : Syntax) : MVarId :=
6811   stx[0].getKind
6812
6813 /--
6814   The elaboration function for `Syntax` created using `mkMVarSyntax`.

```

```

6815  It just converts the metavariable id wrapped by the Syntax into an `Expr`. -/
6816 @[builtinTermElab MVarWithIdKind] def elabMVarWithIdKind : TermElab := fun stx expectedType? =>
6817   return mkInaccessible <| mkMVar (getMVarSyntaxMVarId stx)
6818
6819 @[builtinTermElab inaccessible] def elabInaccessible : TermElab := fun stx expectedType? => do
6820   let e ← elabTerm stx[1] expectedType?
6821   return mkInaccessible e
6822
6823 /-
6824   Patterns define new local variables.
6825   This module collect them and preprocess `__` occurring in patterns.
6826   Recall that an `__` may represent anonymous variables or inaccessible terms
6827   that are implied by typing constraints. Thus, we represent them with fresh named holes `?x`.
6828   After we elaborate the pattern, if the metavariable remains unassigned, we transform it into
6829   a regular pattern variable. Otherwise, it becomes an inaccessible term.
6830
6831   Macros occurring in patterns are expanded before the `collectPatternVars` method is executed.
6832   The following kinds of Syntax are handled by this module
6833   - Constructor applications
6834   - Applications of functions tagged with the `[matchPattern]` attribute
6835   - Identifiers
6836   - Anonymous constructors
6837   - Structure instances
6838   - Inaccessible terms
6839   - Named patterns
6840   - Tuple literals
6841   - Type ascriptions
6842   - Literals: num, string and char
6843 -/
6844 namespace CollectPatternVars
6845
6846 structure State where
6847   found      : NameSet := {}
6848   vars       : Array PatternVar := #[]
6849
6850 abbrev M := StateRefT State TermElabM
6851
6852 private def throwCtorExpected {α} : M α :=
6853   throwError "invalid pattern, constructor or constant marked with '[matchPattern]' expected"
6854
6855 private def getNumExplicitCtorParams (ctorVal : ConstructorVal) : TermElabM Nat :=
6856   forallBoundedTelescope ctorVal.type ctorVal.numParams fun ps _ => do
6857     let mut result := 0
6858     for p in ps do
6859       let localDecl ← getLocalDecl p.fvarId!
6860       if localDecl.binderInfo.isExplicit then
6861         result := result+1

```

```

6862     pure result
6863
6864 private def throwInvalidPattern {α} : M α :=
6865   throwError "invalid pattern"
6866
6867 /-
6868 An application in a pattern can be
6869
6870 1- A constructor application
6871     The elaborator assumes fields are accessible and inductive parameters are not accessible.
6872
6873 2- A regular application `(f ...)` where `f` is tagged with `[matchPattern]`.
6874     The elaborator assumes implicit arguments are not accessible and explicit ones are accessible.
6875 -/
6876
6877 structure Context where
6878   funId      : Syntax
6879   ctorVal?   : Option ConstructorVal -- It is `some`, if constructor application
6880   explicit   : Bool
6881   ellipsis   : Bool
6882   paramDecls : Array (Name × BinderInfo) -- parameters names and binder information
6883   paramDeclIdx : Nat := 0
6884   namedArgs  : Array NamedArg
6885   args       : List Arg
6886   newArgs    : Array Syntax := #[]
6887   deriving   Inhabited
6888
6889 private def isDone (ctx : Context) : Bool :=
6890   ctx.paramDeclIdx ≥ ctx.paramDecls.size
6891
6892 private def finalize (ctx : Context) : M Syntax := do
6893   if ctx.namedArgs.isEmpty && ctx.args.isEmpty then
6894     let fStx ← `(@(ctx.funId):ident)
6895     return Syntax.mkApp fStx ctx.newArgs
6896   else
6897     throwError "too many arguments"
6898
6899 private def isNextArgAccessible (ctx : Context) : Bool :=
6900   let i := ctx.paramDeclIdx
6901   match ctx.ctorVal? with
6902   | some ctorVal => i ≥ ctorVal.numParams -- For constructor applications only fields are accessible
6903   | none =>
6904     if h : i < ctx.paramDecls.size then
6905       -- For `[matchPattern]` applications, only explicit parameters are accessible.
6906       let d := ctx.paramDecls.get {i, h}
6907       d.2.isExplicit
6908     else

```

```

6909     false
6910
6911 private def getNextParam (ctx : Context) : (Name × BinderInfo) × Context :=
6912   let i := ctx.paramDeclIdx
6913   let d := ctx.paramDecls[i]
6914   (d, { ctx with paramDeclIdx := ctx.paramDeclIdx + 1 })
6915
6916 private def processVar (idStx : Syntax) : M Syntax := do
6917   unless idStx.isIdent do
6918     throwErrorAt idStx "identifier expected"
6919   let id := idStx.getId
6920   unless id.eraseMacroScopes.isAtomic do
6921     throwError "invalid pattern variable, must be atomic"
6922   if (← get).found.contains id then
6923     throwError "invalid pattern, variable '{id}' occurred more than once"
6924   modify fun s => { s with vars := s.vars.push (PatternVar.localVar id), found := s.found.insert id }
6925   return idStx
6926
6927 private def nameToPattern : Name → TermElabM Syntax
6928 | Name.anonymous => `(Name.anonymous)
6929 | Name.str p s _ => do let p ← nameToPattern p; `(Name.str $p $(quote s) _)
6930 | Name.num p n _ => do let p ← nameToPattern p; `(Name.num $p $(quote n) _)
6931
6932 private def quotedNameToPattern (stx : Syntax) : TermElabM Syntax :=
6933   match stx[0].isNameLit? with
6934   | some val => nameToPattern val
6935   | none     => throwIllFormedSyntax
6936
6937 private def doubleQuotedNameToPattern (stx : Syntax) : TermElabM Syntax := do
6938   match stx[1].isNameLit? with
6939   | some val => nameToPattern (← resolveGlobalConstNoOverloadWithInfo stx[1] val)
6940   | none     => throwIllFormedSyntax
6941
6942 partial def collect (stx : Syntax) : M Syntax := withRef stx <| withFreshMacroScope do
6943   let k := stx.getKind
6944   if k == identKind then
6945     processId stx
6946   else if k == ``Lean.Parser.Term.app then
6947     processCtorApp stx
6948   else if k == ``Lean.Parser.Term.anonymousCtor then
6949     let elems ← stx[1].getArgs.mapSepElemsM collect
6950     return stx.setArg 1 <| mkNullNode elems
6951   else if k == ``Lean.Parser.Term.structInst then
6952     /-
6953     ...
6954     leading_parser "{" >> optional (atomic (termParser >> " with "))
6955     >> many1Indent (group (structInstField >> optional ", "))

```

```

6956         >> optional ".."
6957         >> optional (" : " >> termParser)
6958         >> "}"
6959     ...
6960 -/
6961 let withMod := stx[1]
6962 unless withMod.isNone do
6963   throwErrorAt withMod "invalid struct instance pattern, 'with' is not allowed in patterns"
6964 let fields ← stx[2].getArgs.mapM fun p => do
6965   -- p is of the form (group (structInstField >> optional ", "))
6966   let field := p[0]
6967   -- leading_parser structInstLVal >> " := " >> termParser
6968   let newVal ← collect field[2]
6969   let field := field.setArg 2 newVal
6970   pure <| field.setArg 0 field
6971 return stx.setArg 2 <| mkNullNode fields
6972 else if k == ``Lean.Parser.Term.hole then
6973   let r ← mkMVarSyntax
6974   modify fun s => { s with vars := s.vars.push <| PatternVar.anonymousVar <| getMVarSyntaxMVarId r }
6975   return r
6976 else if k == ``Lean.Parser.Term.paren then
6977   let arg := stx[1]
6978   if arg.isNone then
6979     return stx -- `()`
6980   else
6981     let t := arg[0]
6982     let s := arg[1]
6983     if s.isNone || s[0].getKind == ``Lean.Parser.Term.typeAscription then
6984       -- Ignore `s`, since it empty or it is a type ascription
6985       let t ← collect t
6986       let arg := arg.setArg 0 t
6987       return stx.setArg 1 arg
6988     else
6989       -- Tuple literal is a constructor
6990       let t ← collect t
6991       let arg := arg.setArg 0 t
6992       let tupleTail := s[0]
6993       let tupleTailElems := tupleTail[1].getArgs
6994       let tupleTailElems ← tupleTailElems.mapSepElemsM collect
6995       let tupleTail := tupleTail.setArg 1 <| mkNullNode tupleTailElems
6996       let s := s.setArg 0 tupleTail
6997       let arg := arg.setArg 1 s
6998       return stx.setArg 1 arg
6999 else if k == ``Lean.Parser.Term.explicitUniv then
7000   processCtor stx[0]
7001 else if k == ``Lean.Parser.Term.namedPattern then
7002   /- Recall that

```

```

7003     def namedPattern := check... >> trailing_parser "@" >> termParser -/
7004     let id := stx[0]
7005     discard <| processVar id
7006     let pat := stx[2]
7007     let pat ← collect pat
7008     `(_root_.namedPattern $id $pat)
7009   else if k == ``Lean.Parser.Term.inaccessible then
7010     return stx
7011   else if k == strLitKind then
7012     return stx
7013   else if k == numLitKind then
7014     return stx
7015   else if k == scientificLitKind then
7016     return stx
7017   else if k == charLitKind then
7018     return stx
7019   else if k == ``Lean.Parser.Term.quotedName then
7020     /- Quoted names have an elaboration function associated with them, and they will not be macro expanded.
7021        Note that macro expansion is not a good option since it produces a term using the smart constructors `Name.mkStr`, `Name.mkNum`
7022        instead of the constructors `Name.str` and `Name.num` -/
7023     quotedNameToPattern stx
7024   else if k == ``Lean.Parser.Term.doubleQuotedName then
7025     /- Similar to previous case -/
7026     doubleQuotedNameToPattern stx
7027   else if k == choiceKind then
7028     throwError "invalid pattern, notation is ambiguous"
7029   else
7030     throwInvalidPattern
7031
7032 where
7033
7034 processCtorApp (stx : Syntax) : M Syntax := do
7035   let (f, namedArgs, args, ellipsis) ← expandApp stx true
7036   processCtorAppCore f namedArgs args ellipsis
7037
7038 processCtor (stx : Syntax) : M Syntax := do
7039   processCtorAppCore stx #[] #[] false
7040
7041 /- Check whether `stx` is a pattern variable or constructor-like (i.e., constructor or constant tagged with `[matchPattern]` attribute
7042 processId (stx : Syntax) : M Syntax := do
7043   match (← resolveId? stx "pattern") with
7044   | none => processVar stx
7045   | some f => match f with
7046   | Expr.const fName _ =>
7047     match (← getEnv).find? fName with
7048     | some (ConstantInfo.ctorInfo _) => processCtor stx
7049     | some _ =>

```

```

7050         if hasMatchPatternAttribute (← getEnv) fName then
7051             processCtor stx
7052         else
7053             processVar stx
7054     | none => throwCtorExpected
7055     | _ => processVar stx
7056
7057 pushNewArg (accessible : Bool) (ctx : Context) (arg : Arg) : M Context := do
7058     match arg with
7059     | Arg.stx stx =>
7060         let stx ← if accessible then collect stx else pure stx
7061         return { ctx with newArgs := ctx.newArgs.push stx }
7062     | _ => unreachable!
7063
7064 processExplicitArg (accessible : Bool) (ctx : Context) : M Context := do
7065     match ctx.args with
7066     | [] =>
7067         if ctx.ellipsis then
7068             pushNewArg accessible ctx (Arg.stx (← `(_)))
7069         else
7070             throwError "explicit parameter is missing, unused named arguments {ctx.namedArgs.map fun narg => narg.name}"
7071     | arg::args =>
7072         pushNewArg accessible { ctx with args := args } arg
7073
7074 processImplicitArg (accessible : Bool) (ctx : Context) : M Context := do
7075     if ctx.explicit then
7076         processExplicitArg accessible ctx
7077     else
7078         pushNewArg accessible ctx (Arg.stx (← `(_)))
7079
7080 processCtorAppContext (ctx : Context) : M Syntax := do
7081     if isDone ctx then
7082         finalize ctx
7083     else
7084         let accessible := isNextArgAccessible ctx
7085         let (d, ctx) := getNextParam ctx
7086         match ctx.namedArgs.findIdx? fun namedArg => namedArg.name == d.1 with
7087         | some idx =>
7088             let arg := ctx.namedArgs[idx]
7089             let ctx := { ctx with namedArgs := ctx.namedArgs.eraseIdx idx }
7090             let ctx ← pushNewArg accessible ctx arg.val
7091             processCtorAppContext ctx
7092         | none =>
7093             let ctx ← match d.2 with
7094             | BinderInfo.implicit      => processImplicitArg accessible ctx
7095             | BinderInfo.instImplicit => processImplicitArg accessible ctx
7096             | _                       => processExplicitArg accessible ctx

```

```

7097     processCtorAppContext ctx
7098
7099 processCtorAppCore (f : Syntax) (namedArgs : Array NamedArg) (args : Array Arg) (ellipsis : Bool) : M Syntax := do
7100   let args := args.toList
7101   let (fId, explicit) ← match f with
7102   | `($fId:ident) => pure (fId, false)
7103   | `(@$fId:ident) => pure (fId, true)
7104   | _ => throwError "identifier expected"
7105   let some (Expr.const fName _) ← resolveId? fId "pattern" | throwErrorExpected
7106   let fInfo ← getConstInfo fName
7107   let paramDecls ← forallTelescopeReducing fInfo.type fun xs _ => xs.mapM fun x => do
7108     let d ← getFVarLocalDecl x
7109     return (d.userName, d.binderInfo)
7110   match fInfo with
7111   | ConstantInfo.ctorInfo val =>
7112     processCtorAppContext
7113       { funId := fId, explicit := explicit, ctorVal? := val, paramDecls := paramDecls, namedArgs := namedArgs, args := args, ellipsis
7114       | _ =>
7115         if hasMatchPatternAttribute (← getEnv) fName then
7116           processCtorAppContext
7117             { funId := fId, explicit := explicit, ctorVal? := none, paramDecls := paramDecls, namedArgs := namedArgs, args := args, ellipsis
7118           else
7119             throwErrorExpected
7120
7121 def main (alt : MatchAltView) : M MatchAltView := do
7122   let patterns ← alt.patterns.mapM fun p => do
7123     trace[Elab.match] "collecting variables at pattern: {p}"
7124     collect p
7125   return { alt with patterns := patterns }
7126
7127 end CollectPatternVars
7128
7129 private def collectPatternVars (alt : MatchAltView) : TermElabM (Array PatternVar × MatchAltView) := do
7130   let (alt, s) ← (CollectPatternVars.main alt).run {}
7131   return (s.vars, alt)
7132
7133 /- Return the pattern variables in the given pattern.
7134 Remark: this method is not used here, but in other macros (e.g., at `Do.lean`). -/
7135 def getPatternVars (patternStx : Syntax) : TermElabM (Array PatternVar) := do
7136   let patternStx ← liftMacroM <| expandMacros patternStx
7137   let (_, s) ← (CollectPatternVars.collect patternStx).run {}
7138   return s.vars
7139
7140 def getPatternsVars (patterns : Array Syntax) : TermElabM (Array PatternVar) := do
7141   let collect : CollectPatternVars.M Unit := do
7142     for pattern in patterns do
7143       discard <| CollectPatternVars.collect (← liftMacroM <| expandMacros pattern)

```



```

7144 let (_, s) ← collect.run {}
7145 return s.vars
7146
7147 /- We convert the collected `PatternVar`s into `PatternVarDecl` -/
7148 inductive PatternVarDecl where
7149   /- For `anonymousVar`, we create both a metavariable and a free variable. The free variable is used as an assignment for the metavariable
7150   when it is not assigned during pattern elaboration. -/
7151   | anonymousVar (mvarId : MVarId) (fvarId : FVarId)
7152   | localVar    (fvarId : FVarId)
7153
7154 private partial def withPatternVars {α} (pVars : Array PatternVar) (k : Array PatternVarDecl → TermElabM α) : TermElabM α :=
7155   let rec loop (i : Nat) (decls : Array PatternVarDecl) := do
7156     if h : i < pVars.size then
7157       match pVars.get (i, h) with
7158       | PatternVar.anonymousVar mvarId =>
7159         let type ← mkFreshTypeMVar
7160         let userName ← mkFreshBinderName
7161         withLocalDecl userName BinderInfo.default type fun x =>
7162           loop (i+1) (decls.push (PatternVarDecl.anonymousVar mvarId x.fvarId!))
7163       | PatternVar.localVar userName =>
7164         let type ← mkFreshTypeMVar
7165         withLocalDecl userName BinderInfo.default type fun x =>
7166           loop (i+1) (decls.push (PatternVarDecl.localVar x.fvarId!))
7167     else
7168       /- We must create the metavariables for `PatternVar.anonymousVar` AFTER we create the new local decls using `withLocalDecl`.
7169       Reason: their scope must include the new local decls since some of them are assigned by typing constraints. -/
7170       decls.forM fun decl => match decl with
7171       | PatternVarDecl.anonymousVar mvarId fvarId => do
7172         let type ← inferType (mkFVar fvarId)
7173         discard <| mkFreshExprMVarWithId mvarId type
7174       | _ => pure ()
7175   k decls
7176   loop 0 #[]
7177
7178 /-
7179 Remark: when performing dependent pattern matching, we often had to write code such as
7180
7181 ```lean
7182 def Vec.map' (f : α → β) (xs : Vec α n) : Vec β n :=
7183   match n, xs with
7184   | _, nil      => nil
7185   | _, cons a as => cons (f a) (map' f as)
7186 ```
7187 We had to include `n` and the `'_`s because the type of `xs` depends on `n`.
7188 Moreover, `nil` and `cons a as` have different types.
7189 This was quite tedious. So, we have implemented an automatic "discriminant refinement procedure".
7190 The procedure is based on the observation that we get a type error whenever we forget to include `'_`s

```

```

7191 and the indices a discriminant depends on. So, we catch the exception, check whether the type of the discriminant
7192 is an indexed family, and add their indices as new discriminants.
7193
7194 The current implementation, adds indices as they are found, and does not
7195 try to "sort" the new discriminants.
7196
7197 If the refinement process fails, we report the original error message.
7198 -/
7199
7200 /- Auxiliary structure for storing a type mismatch exception when processing the
7201     pattern #`idx` of some alternative. -/
7202 structure PatternElabException where
7203   ex : Exception
7204   idx : Nat
7205
7206 private def elabPatterns (patternStxs : Array Syntax) (matchType : Expr) : ExceptT PatternElabException TermElabM (Array Expr × Expr) :=
7207   withReader (fun ctx => { ctx with implicitLambda := false }) do
7208     let mut patterns := #[]
7209     let mut matchType := matchType
7210     for idx in [:patternStxs.size] do
7211       let patternStx := patternStxs[idx]
7212       matchType ← whnf matchType
7213       match matchType with
7214       | Expr.forallE _ d b _ =>
7215         let pattern ←
7216           try
7217             liftM <| withSynthesize <| withoutErrToSorry <| elabTermEnsuringType patternStx d
7218           catch ex =>
7219             -- Wrap the type mismatch exception for the "discriminant refinement" feature.
7220             throwThe PatternElabException { ex := ex, idx := idx }
7221         matchType := b.instantiate1 pattern
7222         patterns := patterns.push pattern
7223       | _ => throwError "unexpected match type"
7224     return (patterns, matchType)
7225
7226 def finalizePatternDecls (patternVarDecls : Array PatternVarDecl) : TermElabM (Array LocalDecl) := do
7227   let mut decls := #[]
7228   for pdecl in patternVarDecls do
7229     match pdecl with
7230     | PatternVarDecl.localVar fvarId =>
7231       let decl ← getLocalDecl fvarId
7232       let decl ← instantiateLocalDeclMVars decl
7233       decls := decls.push decl
7234     | PatternVarDecl.anonymousVar mvarId fvarId =>
7235       let e ← instantiateMVars (mkMVar mvarId);
7236       trace[Elab.match] "finalizePatternDecls: mvarId: {mvarId} := {e}, fvar: {mkFVar fvarId}"
7237       match e with

```

```

7238 | Expr.mvar newMVarId _ =>
7239 |/- Metavariable was not assigned, or assigned to another metavariable. So,
7240 |   we assign to the auxiliary free variable we created at `withPatternVars` to `newMVarId`. -/
7241 |   assignExprMVar newMVarId (mkFVar fvarId)
7242 |   trace[Elab.match] "finalizePatternDecls: {mkMVar newMVarId} := {mkFVar fvarId}"
7243 |   let decl ← getLocalDecl fvarId
7244 |   let decl ← instantiateLocalDeclMVars decl
7245 |   decls := decls.push decl
7246 | _ => pure ()
7247 |/- We perform a topological sort (dependencies) on `decls` because the pattern elaboration process may produce a sequence where a decla
7248 | sortLocalDecls decls
7249
7250 open Meta.Match (Pattern Pattern.var Pattern.inaccessible Pattern.ctor Pattern.as Pattern.val Pattern.arrayLit AltLHS MatcherResult)
7251
7252 namespace ToDepElimPattern
7253
7254 structure State where
7255   found      : NameSet := {}
7256   localDecls : Array LocalDecl
7257   newLocals  : NameSet := {}
7258
7259 abbrev M := StateRefT State TermElabM
7260
7261 private def alreadyVisited (fvarId : FVarId) : M Bool := do
7262   let s ← get
7263   return s.found.contains fvarId
7264
7265 private def markAsVisited (fvarId : FVarId) : M Unit :=
7266   modify fun s => { s with found := s.found.insert fvarId }
7267
7268 private def throwInvalidPattern {α} (e : Expr) : M α :=
7269   throwError "invalid pattern {indentExpr e}"
7270
7271 |/- Create a new LocalDecl `x` for the metavariable `mvar`, and return `Pattern.var x` -/
7272 private def mkLocalDeclFor (mvar : Expr) : M Pattern := do
7273   let mvarId := mvar.mvarId!
7274   let s ← get
7275   match (← getExprMVarAssignment? mvarId) with
7276   | some val => return Pattern.inaccessible val
7277   | none =>
7278     let fvarId ← mkFreshId
7279     let type ← inferType mvar
7280     |/- HACK: `fvarId` is not in the scope of `mvarId`
7281     |/- If this generates problems in the future, we should update the metavariable declarations. -/
7282     assignExprMVar mvarId (mkFVar fvarId)
7283     let userName ← mkFreshBinderName
7284     let newDecl := LocalDecl.cdecl arbitrary fvarId userName type BinderInfo.default;

```

```

7285     modify fun s =>
7286       { s with
7287         newLocals := s.newLocals.insert fvarId,
7288         localDecls :=
7289           match s.localDecls.findIdx? fun decl => mvar.occurs decl.type with
7290           | none   => s.localDecls.push newDecl -- None of the existing declarations depend on `mvar`
7291           | some i => s.localDecls.insertAt i newDecl }
7292     return Pattern.var fvarId
7293
7294 partial def main (e : Expr) : M Pattern := do
7295   let isLocalDecl (fvarId : FVarId) : M Bool := do
7296     return (← get).localDecls.any fun d => d.fvarId == fvarId
7297   let mkPatternVar (fvarId : FVarId) (e : Expr) : M Pattern := do
7298     if (← alreadyVisited fvarId) then
7299       return Pattern.inaccessible e
7300     else
7301       markAsVisited fvarId
7302       return Pattern.var e.fvarId!
7303   let mkInaccessible (e : Expr) : M Pattern := do
7304     match e with
7305     | Expr.fvar fvarId _ =>
7306       if (← isLocalDecl fvarId) then
7307         mkPatternVar fvarId e
7308       else
7309         return Pattern.inaccessible e
7310     | _ =>
7311       return Pattern.inaccessible e
7312   match inaccessible? e with
7313   | some t => mkInaccessible t
7314   | none =>
7315     match e.arrayLit? with
7316     | some (α, lits) =>
7317       return Pattern.arrayLit α (← lits.mapM main)
7318     | none =>
7319       if e.isAppOfAry `namedPattern 3 then
7320         let p ← main <| e.getArg! 2
7321         match e.getArg! 1 with
7322         | Expr.fvar fvarId _ => return Pattern.as fvarId p
7323         | _                  => throwError "unexpected occurrence of auxiliary declaration 'namedPattern'"
7324       else if e.isNatLit || e.isStringLit || e.isCharLit then
7325         return Pattern.val e
7326       else if e.isFVar then
7327         let fvarId := e.fvarId!
7328         unless (← isLocalDecl fvarId) do
7329           throwInvalidPattern e
7330         mkPatternVar fvarId e
7331       else if e.isMVar then

```

```

7332     mkLocalDeclFor e
7333   else
7334     let newE ← whnf e
7335     if newE != e then
7336       main newE
7337     else matchConstCtor e.getAppFn (fun _ => throwInvalidPattern e) fun v us => do
7338       let args := e.getAppArgs
7339       unless args.size == v.numParams + v.numFields do
7340         throwInvalidPattern e
7341       let params := args.extract 0 v.numParams
7342       let fields := args.extract v.numParams args.size
7343       let fields ← fields.mapM main
7344       return Pattern.ctor v.name us params.toList fields.toList
7345
7346 end ToDepElimPattern
7347
7348 def withDepElimPatterns {α} (localDecls : Array LocalDecl) (ps : Array Expr) (k : Array LocalDecl → Array Pattern → TermElabM α) : TermE
7349   let (patterns, s) ← (ps.mapM ToDepElimPattern.main).run { localDecls := localDecls }
7350   let localDecls ← s.localDecls.mapM fun d => instantiateLocalDeclMVars d
7351   /- toDepElimPatterns may have added new localDecls. Thus, we must update the local context before we execute `k` -/
7352   let lctx ← getLCtx
7353   let lctx := localDecls.foldl (fun (lctx : LocalContext) d => lctx.erase d.fvarId) lctx
7354   let lctx := localDecls.foldl (fun (lctx : LocalContext) d => lctx.addDecl d) lctx
7355   withTheReader Meta.Context (fun ctx => { ctx with lctx := lctx }) do
7356     k localDecls patterns
7357
7358 private def withElaboratedLHS {α} (ref : Syntax) (patternVarDecls : Array PatternVarDecl) (patternStxs : Array Syntax) (matchType : Expr
7359   (k : AltLHS → Expr → TermElabM α) : ExceptT PatternElabException TermElabM α := do
7360   let (patterns, matchType) ← withSynthesize <| elabPatterns patternStxs matchType
7361   id (α := TermElabM α) do
7362     let localDecls ← finalizePatternDecls patternVarDecls
7363     let patterns ← patterns.mapM (instantiateMVars ·)
7364     withDepElimPatterns localDecls patterns fun localDecls patterns =>
7365       k { ref := ref, fvarDecls := localDecls.toList, patterns := patterns.toList } matchType
7366
7367 private def elabMatchAltView (alt : MatchAltView) (matchType : Expr) : ExceptT PatternElabException TermElabM (AltLHS × Expr) := withRef
7368   let (patternVars, alt) ← collectPatternVars alt
7369   trace[Elab.match] "patternVars: {patternVars}"
7370   withPatternVars patternVars fun patternVarDecls => do
7371     withElaboratedLHS alt.ref patternVarDecls alt.patterns matchType fun altLHS matchType => do
7372       let rhs ← elabTermEnsuringType alt.rhs matchType
7373       let xs := altLHS.fvarDecls.toArray.map LocalDecl.toExpr
7374       let rhs ← if xs.isEmpty then pure <| mkSimpleThunk rhs else mkLambdaFVars xs rhs
7375       trace[Elab.match] "rhs: {rhs}"
7376       return (altLHS, rhs)
7377
7378 /--

```

```

7379 Collect indices for the "discriminant refinement feature". This method is invoked
7380 when we detect a type mismatch at a pattern #`idx` of some alternative. -/
7381 private def getIndicesToInclude (discrs : Array Expr) (idx : Nat) : TermElabM (Array Expr) := do
7382   let discrType ← whnfD (← inferType discrs[idx])
7383   matchConstInduct discrType.getAppFn (fun _ => return #[]) fun info _ => do
7384     let mut result := #[]
7385     let args := discrType.getAppArgs
7386     for arg in args[info.numParams : args.size] do
7387       unless (← discrs.anyM fun discr => isDefEq discr arg) do
7388         result := result.push arg
7389     return result
7390
7391 private partial def elabMatchAltViews (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) : TermElabM (Array Expr ×
7392   loop discrs matchType altViews none
7393 where
7394   /-
7395     "Discriminant refinement" main loop.
7396     `first?` contains the first error message we found before updated the `discrs`. -/
7397   loop (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) (first? : Option (SavedState × Exception))
7398     : TermElabM (Array Expr × Expr × Array (AltLHS × Expr) × Bool) := do
7399     let s ← saveState
7400     match ← altViews.mapM (fun alt => elabMatchAltView alt matchType) |>.run with
7401     | Except.ok alts => return (discrs, matchType, alts, first?.isSome)
7402     | Except.error { idx := idx, ex := ex } =>
7403       let indices ← getIndicesToInclude discrs idx
7404       if indices.isEmpty then
7405         throwEx (← updateFirst first? ex)
7406       else
7407         let first ← updateFirst first? ex
7408         s.restore
7409         let indices ← collectDeps indices discrs
7410         let matchType ←
7411           try
7412             updateMatchType indices matchType
7413           catch ex =>
7414             throwEx first
7415         let altViews ← addWildcardPatterns indices.size altViews
7416         let discrs := indices ++ discrs
7417         loop discrs matchType altViews first
7418
7419   throwEx {α} (p : SavedState × Exception) : TermElabM α := do
7420     p.1.restore; throw p.2
7421
7422   updateFirst (first? : Option (SavedState × Exception)) (ex : Exception) : TermElabM (SavedState × Exception) := do
7423     match first? with
7424     | none => return (← saveState, ex)
7425     | some first => return first

```

```

7426
7427 containsFVar (es : Array Expr) (fvarId : FVarId) : Bool :=
7428   es.any fun e => e.isFVar && e.fvarId! == fvarId
7429
7430 /- Update `indices` by including any free variable `x` s.t.
7431    - Type of some `discr` depends on `x`.
7432    - Type of `x` depends on some free variable in `indices`.
7433
7434    If we don't include these extra variables in indices, then
7435    `updateMatchType` will generate a type incorrect term.
7436    For example, suppose `discr` contains `h : @HEq α a α b`, and
7437    `indices` is `#[α, b]`, and `matchType` is `@HEq α a α b → B`.
7438    `updateMatchType indices matchType` produces the type
7439    `(α' : Type) → (b : α') → @HEq α' a α' b → B` which is type incorrect
7440    because we have `a : α`.
7441    The method `collectDeps` will include `a` into `indices`.
7442
7443    This method does not handle dependencies among non-free variables.
7444    We rely on the type checking method `check` at `updateMatchType`. -/
7445 collectDeps (indices : Array Expr) (discrs : Array Expr) : TermElabM (Array Expr) := do
7446   let mut s : CollectFVars.State := {}
7447   for discr in discrs do
7448     s := collectFVars s (← instantiateMVars (← inferType discr))
7449   let (indicesFVar, indicesNonFVar) := indices.split Expr.isFVar
7450   let indicesFVar := indicesFVar.map Expr.fvarId!
7451   let mut toAdd := #[]
7452   for fvarId in s.fvarSet.toList do
7453     unless containsFVar discrs fvarId || containsFVar indices fvarId do
7454       let localDecl ← getLocalDecl fvarId
7455       let mctx ← getMCtx
7456       for indexFVarId in indicesFVar do
7457         if mctx.localDeclDependsOn localDecl indexFVarId then
7458           toAdd := toAdd.push fvarId
7459   let lctx ← getLCtx
7460   let indicesFVar := (indicesFVar ++ toAdd).qsort fun fvarId₁ fvarId₂ =>
7461     (lctx.get! fvarId₁).index < (lctx.get! fvarId₂).index
7462   return indicesFVar.map mkFVar ++ indicesNonFVar
7463
7464 updateMatchType (indices : Array Expr) (matchType : Expr) : TermElabM Expr := do
7465   let matchType ← indices.foldrM (init := matchType) fun index matchType => do
7466     let indexType ← inferType index
7467     let matchTypeBody ← kabstract matchType index
7468     let userName ← mkUserNameFor index
7469     return Lean.mkForall userName BinderInfo.default indexType matchTypeBody
7470   check matchType
7471   return matchType
7472

```

```

7473 addWildcardPatterns (num : Nat) (altViews : Array MatchAltView) : TermElabM (Array MatchAltView) := do
7474   let hole := mkHole (← getRef)
7475   let wildcards := mkArray num hole
7476   return altViews.map fun altView => { altView with patterns := wildcards ++ altView.patterns }
7477
7478 def mkMatcher (elimName : Name) (matchType : Expr) (numDiscrs : Nat) (lhss : List AltLHS) : TermElabM MatcherResult :=
7479   Meta.Match.mkMatcher elimName matchType numDiscrs lhss
7480
7481 register_builtin_option match.ignoreUnusedAlts : Bool := {
7482   defValue := false
7483   descr := "if true, do not generate error if an alternative is not used"
7484 }
7485
7486 def reportMatcherResultErrors (altLHSS : List AltLHS) (result : MatcherResult) : TermElabM Unit := do
7487   unless result.counterExamples.isEmpty do
7488     withHeadRefOnly <| throwError "missing cases:\n{Meta.Match.counterExamplesToMessageData result.counterExamples}"
7489   unless match.ignoreUnusedAlts.get (← getOptions) || result.unusedAltIdxs.isEmpty do
7490     let mut i := 0
7491     for alt in altLHSS do
7492       if result.unusedAltIdxs.contains i then
7493         withRef alt.ref do
7494           logError "redundant alternative"
7495       i := i + 1
7496
7497 /--
7498   If `altLHSS + rhss` is encoding `| PUnit.unit => rhs[0]`, return `rhs[0]`
7499   Otherwise, return none.
7500 -/
7501 private def isMatchUnit? (altLHSS : List Match.AltLHS) (rhss : Array Expr) : MetaM (Option Expr) := do
7502   assert! altLHSS.length == rhss.size
7503   match altLHSS with
7504   | [ { fvarDecls := [], patterns := [ Pattern.ctor `PUnit.unit .. ], .. } ] =>
7505     /- Recall that for alternatives of the form `| PUnit.unit => rhs`, `rhss[0]` is of the form `fun _ : Unit => b`. -/
7506     match rhss[0] with
7507     | Expr.lam _ _ b _ => return if b.hasLooseBVars then none else b
7508     | _ => return none
7509   | _ => return none
7510
7511 /--
7512   "Generalize" variables that depend on the discriminants.
7513
7514   Remarks and limitations:
7515   - If `matchType` is a proposition, then we generalize even when the user did not provide `(generalizing := true)`.
7516     Motivation: users should have control about the actual `match`-expressions in their programs.
7517   - We currently do not generalize let-decls.
7518   - We abort generalization if the new `matchType` is type incorrect.
7519   - Only discriminants that are free variables are considered during specialization.

```



```

7520 - We "generalize" by adding new discriminants and pattern variables. We do not "clear" the generalized variables,
7521 but they become inaccessible since they are shadowed by the patterns variables. We assume this is ok since
7522 this is the exact behavior users would get if they had written it by hand. Recall there is no `clear` in term mode.
7523 -/
7524 private def generalize (discrs : Array Expr) (matchType : Expr) (altViews : Array MatchAltView) (generalizing? : Option Bool) : TermElab
7525   let gen ←
7526     match generalizing? with
7527     | some g => pure g
7528     | _ => isProp matchType
7529   if !gen then
7530     return (discrs, matchType, altViews, false)
7531   else
7532     let ysFVarIds ← getFVarsToGeneralize discrs
7533     /- let-decls are currently being ignored by the generalizer. -/
7534     let ysFVarIds ← ysFVarIds.filterM fun fvarId => return !(← getLocalDecl fvarId).isLet
7535     if ysFVarIds.isEmpty then
7536       return (discrs, matchType, altViews, false)
7537     else
7538       let ys := ysFVarIds.map mkFVar
7539       -- trace[Meta.debug] "ys: {ys}, discrs: {discrs}"
7540       let matchType' ← forallBoundedTelescope matchType discrs.size fun ds type => do
7541         let type ← mkForallFVars ys type
7542         let (discrs', ds') := Array.unzip <| Array.zip discrs ds |>.filter fun (di, d) => di.isFVar
7543         let type := type.replaceFVars discrs' ds'
7544         mkForallFVars ds type
7545       -- trace[Meta.debug] "matchType': {matchType'}"
7546       if (← isTypeCorrect matchType') then
7547         let discrs := discrs ++ ys
7548         let altViews ← altViews.mapM fun altView => do
7549           let patternVars ← getPatternsVars altView.patterns
7550           -- We traverse backwards because we want to keep the most recent names.
7551           -- For example, if `ys` contains `#[h, h]`, we want to make sure `mkFreshUsername` is applied to the first `h`,
7552           -- since it is already shadowed by the second.
7553           let ysUserNames ← ys.foldrM (init := #[]) fun ys ysUserNames => do
7554             let yDecl ← getLocalDecl ys.fvarId!
7555             let mut yUserName := yDecl.userName
7556             if ysUserNames.contains yUserName then
7557               yUserName ← mkFreshUsername yUserName
7558             -- Explicitly provided pattern variables shadow `y`
7559             else if patternVars.any fun | PatternVar.localVar x => x == yUserName | _ => false then
7560               yUserName ← mkFreshUsername yUserName
7561             return ysUserNames.push yUserName
7562           let ysIds ← ysUserNames.reverse.mapM fun n => return mkIdentFrom (← getRef) n
7563           return { altView with patterns := altView.patterns ++ ysIds }
7564         return (discrs, matchType', altViews, true)
7565       else
7566         return (discrs, matchType, altViews, true)

```

```

7567
7568 private def elabMatchAux (generalizing? : Option Bool) (discrStxs : Array Syntax) (altViews : Array MatchAltView) (matchOptType : Syntax
7569   : TermElabM Expr := do
7570   let mut generalizing? := generalizing?
7571   if !matchOptType.isNone then
7572     if generalizing? == some true then
7573       throwError "the '(generalizing := true)' parameter is not supported when the 'match' type is explicitly provided"
7574     generalizing? := some false
7575   let (discrs, matchType, altLHSS, isDep, rhss) ← commitIfDidNotPostpone do
7576     let (discrs, matchType, isDep, altViews) ← elabMatchTypeAndDiscrs discrStxs matchOptType altViews expectedType
7577     let (discrs, matchType, altViews, gen) ← generalize discrs matchType altViews generalizing?
7578     let isDep := isDep || gen
7579     let matchAlts ← liftMacroM <| expandMacrosInPatterns altViews
7580     trace[Elab.match] "matchType: {matchType}"
7581     let (discrs, matchType, alts, refined) ← elabMatchAltViews discrs matchType matchAlts
7582     let isDep := isDep || refined
7583   /-
7584     We should not use `synthesizeSyntheticMVarsNoPostponing` here. Otherwise, we will not be
7585     able to elaborate examples such as:
7586     ```
7587     def f (x : Nat) : Option Nat := none
7588
7589     def g (xs : List (Nat × Nat)) : IO Unit :=
7590       xs.forM fun x =>
7591         match f x.fst with
7592         | _ => pure ()
7593     ```
7594     If `synthesizeSyntheticMVarsNoPostponing`, the example above fails at `x.fst` because
7595     the type of `x` is only available after we proces the last argument of `List.forM`.
7596
7597     We apply pending default types to make sure we can process examples such as
7598     ```
7599     let (a, b) := (0, 0)
7600     ```
7601   -/
7602   synthesizeSyntheticMVarsUsingDefault
7603   let rhss := alts.map Prod.snd
7604   let matchType ← instantiateMVars matchType
7605   let altLHSS ← alts.toList.mapM fun alt => do
7606     let altLHS ← Match.instantiateAltLHSMVars alt.1
7607     /- Remark: we try to postpone before throwing an error.
7608        The combinator `commitIfDidNotPostpone` ensures we backtrack any updates that have been performed.
7609        The quick-check `waitExpectedTypeAndDiscrs` minimizes the number of scenarios where we have to postpone here.
7610        Here is an example that passes the `waitExpectedTypeAndDiscrs` test, but postpones here.
7611        ```
7612        def bad (ps : Array (Nat × Nat)) : Array (Nat × Nat) :=
7613          (ps.filter fun (p : Prod _) =>

```

```

7614         match p with
7615         | (x, y) => x == 0)
7616     ++
7617     `ps
7618     ``
7619     When we try to elaborate `fun (p : Prod _ _) => ...` for the first time, we haven't propagated the type of `ps` yet
7620     because `Array.filter` has type `{α : Type u_1} → (α → Bool) → (as : Array α) → optParam Nat 0 → optParam Nat (Array.size as) →
7621     However, the partial type annotation `(p : Prod _ _)` makes sure we succeed at the quick-check `waitExpectedTypeAndDiscrs`.
7622     -/
7623     withRef altLHS.ref do
7624         for d in altLHS.fvarDecls do
7625             if d.hasExprMVar then
7626                 withExistingLocalDecls altLHS.fvarDecls do
7627                     tryPostpone
7628                     throwMVarError m!"invalid match-expression, type of pattern variable '{d.toExpr}' contains metavariables{indentExpr d.type
7629         for p in altLHS.patterns do
7630             if p.hasExprMVar then
7631                 withExistingLocalDecls altLHS.fvarDecls do
7632                     tryPostpone
7633                     throwMVarError m!"invalid match-expression, pattern contains metavariables{indentExpr (← p.toExpr)}"
7634         pure altLHS
7635     return (discrs, matchType, altLHSS, isDep, rhss)
7636 if let some r ← if isDep then pure none else isMatchUnit? altLHSS rhss then
7637     return r
7638 else
7639     let numDiscrs := discrs.size
7640     let matcherName ← mkAuxName `match
7641     let matcherResult ← mkMatcher matcherName matchType numDiscrs altLHSS
7642     let motive ← forallBoundedTelescope matchType numDiscrs fun xs matchType => mkLambdaFVars xs matchType
7643     reportMatcherResultErrors altLHSS matcherResult
7644     let r := mkApp matcherResult.matcher motive
7645     let r := mkAppN r discrs
7646     let r := mkAppN r rhss
7647     trace[Elab.match] "result: {r}"
7648     return r
7649
7650 private def getDiscrs (matchStx : Syntax) : Array Syntax :=
7651     matchStx[2].getSepArgs
7652
7653 private def getMatchOptType (matchStx : Syntax) : Syntax :=
7654     matchStx[3]
7655
7656 private def expandNonAtomicDiscrs? (matchStx : Syntax) : TermElabM (Option Syntax) :=
7657     let matchOptType := getMatchOptType matchStx;
7658     if matchOptType.isNone then do
7659         let discrs := getDiscrs matchStx;
7660         let allLocal ← discrs.allM fun discr => Option.isSome <$> isAtomicDiscr? discr[1]

```

```

7661   if allLocal then
7662     return none
7663   else
7664     -- We use `foundFVars` to make sure the discriminants are distinct variables.
7665     -- See: code for computing "matchType" at `elabMatchTypeAndDiscrs`
7666     let rec loop (discrs : List Syntax) (discrsNew : Array Syntax) (foundFVars : NameSet) := do
7667       match discrs with
7668       | [] =>
7669         let discrs := Syntax.mkSep discrsNew (mkAtomFrom matchStx ", ");
7670         pure (matchStx.setArg 2 discrs)
7671       | discr :: discrs =>
7672         -- Recall that
7673         -- matchDiscr := leading_parser optional (ident >> ":") >> termParser
7674         let term := discr[1]
7675         let addAux : TermElabM Syntax := withFreshMacroScope do
7676           let d ← `(_discr);
7677           unless isAuxDiscrName d.getId do -- Use assertion?
7678             throwError "unexpected internal auxiliary discriminant name"
7679           let discrNew := discr.setArg 1 d;
7680           let r ← loop discrs (discrsNew.push discrNew) foundFVars
7681           `(let _discr := $term; $r)
7682         match (← isAtomicDiscr? term) with
7683         | some x => if x.isFVar then loop discrs (discrsNew.push discr) (foundFVars.insert x.fvarId!) else addAux
7684         | none   => addAux
7685     return some (← loop discrs.toList #[] {})
7686   else
7687     -- We do not pull non atomic discriminants when match type is provided explicitly by the user
7688     return none
7689
7690 private def waitExpectedType (expectedType? : Option Expr) : TermElabM Expr := do
7691   tryPostponeIfNoneOrMVar expectedType?
7692   match expectedType? with
7693   | some expectedType => pure expectedType
7694   | none              => mkFreshTypeMVar
7695
7696 private def tryPostponeIfDiscrTypeIsMVar (matchStx : Syntax) : TermElabM Unit := do
7697   -- We don't wait for the discriminants types when match type is provided by user
7698   if getMatchOptType matchStx |>.isNone then
7699     let discrs := getDiscrs matchStx
7700     for discr in discrs do
7701       let term := discr[1]
7702       match (← isAtomicDiscr? term) with
7703       | none   => throwErrorAt discr "unexpected discriminant" -- see `expandNonAtomicDiscrs?`
7704       | some d =>
7705         let dType ← inferType d
7706         trace[Elab.match] "discr {d} : {dType}"
7707         tryPostponeIfMVar dType

```

```

7708
7709 /-
7710 We (try to) elaborate a `match` only when the expected type is available.
7711 If the `matchType` has not been provided by the user, we also try to postpone elaboration if the type
7712 of a discriminant is not available. That is, it is of the form `(?m ...)` .
7713 We use `expandNonAtomicDiscrs?` to make sure all discriminants are local variables.
7714 This is a standard trick we use in the elaborator, and it is also used to elaborate structure instances.
7715 Suppose, we are trying to elaborate
7716 ```
7717 match g x with
7718 | ... => ...
7719 ```
7720 `expandNonAtomicDiscrs?` converts it intro
7721 ```
7722 let _discr := g x
7723 match _discr with
7724 | ... => ...
7725 ```
7726 Thus, at `tryPostponeIfDiscrTypeIsMVar` we only need to check whether the type of `_discr` is not of the form `(?m ...)` .
7727 Note that, the auxiliary variable `_discr` is expanded at `elabAtomicDiscr`.
7728
7729 This elaboration technique is needed to elaborate terms such as:
7730 ```lean
7731 xs.filter fun (a, b) => a > b
7732 ```
7733 which are syntax sugar for
7734 ```lean
7735 List.filter (fun p => match p with | (a, b) => a > b) xs
7736 ```
7737 When we visit `match p with | (a, b) => a > b`, we don't know the type of `p` yet.
7738 -/
7739 private def waitExpectedTypeAndDiscrs (matchStx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
7740   tryPostponeIfNoneOrMVar expectedType?
7741   tryPostponeIfDiscrTypeIsMVar matchStx
7742   match expectedType? with
7743   | some expectedType => return expectedType
7744   | none              => mkFreshTypeMVar
7745
7746 /-
7747 ```
7748 leading_parser:leadPrec "match " >> sepBy1 matchDiscr ", " >> optType >> " with " >> matchAlts
7749 ```
7750 Remark the `optIdent` must be `none` at `matchDiscr`. They are expanded by `expandMatchDiscr?`.
7751 -/
7752 private def elabMatchCore (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
7753   let expectedType ← waitExpectedTypeAndDiscrs stx expectedType?
7754   let discrStxs := (getDiscrs stx).map fun d => d

```

```

7755 let gen?      := getMatchGeneralizing? stx
7756 let altViews   := getMatchAlts stx
7757 let matchOptType := getMatchOptType stx
7758 elabMatchAux gen? discrStxs altViews matchOptType expectedType
7759
7760 private def isPatternVar (stx : Syntax) : TermElabM Bool := do
7761   match (← resolveId? stx "pattern") with
7762   | none   => isAtomicIdent stx
7763   | some f => match f with
7764   | Expr.const fName _ _ =>
7765     match (← getEnv).find? fName with
7766     | some (ConstantInfo.ctorInfo _) => return false
7767     | some _                        => return !hasMatchPatternAttribute (← getEnv) fName
7768     | _                             => isAtomicIdent stx
7769   | _ => isAtomicIdent stx
7770 where
7771   isAtomicIdent (stx : Syntax) : Bool :=
7772     stx.isIdent && stx.getId.eraseMacroScopes.isAtomic
7773
7774 -- leading_parser "match " >> sepBy1 termParser ", " >> optType >> " with " >> matchAlts
7775 @[builtinTermElab «match»] def elabMatch : TermElab := fun stx expectedType? => do
7776   match stx with
7777   | `(match $discr:term with | $y:ident => $rhs:term) =>
7778     if (← isPatternVar y) then expandSimpleMatch stx discr y rhs expectedType? else elabMatchDefault stx expectedType?
7779   | _ => elabMatchDefault stx expectedType?
7780 where
7781   elabMatchDefault (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
7782     match (← expandNonAtomicDiscrs? stx) with
7783     | some stxNew => withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
7784     | none =>
7785       let discrs      := getDiscrs stx;
7786       let matchOptType := getMatchOptType stx;
7787       if !matchOptType.isNone && discrs.any fun d => !d[0].isNone then
7788         throwErrorAt matchOptType "match expected type should not be provided when discriminants with equality proofs are used"
7789       elabMatchCore stx expectedType?
7790
7791 builtin_initialize
7792   registerTraceClass `Elab.match
7793
7794 -- leading_parser:leadPrec "nomatch " >> termParser
7795 @[builtinTermElab «nomatch»] def elabNoMatch : TermElab := fun stx expectedType? => do
7796   match stx with
7797   | `(nomatch $discrExpr) =>
7798     match ← isLocalIdent? discrExpr with
7799     | some _ =>
7800       let expectedType ← waitExpectedType expectedType?
7801       let discr := Syntax.node ``Lean.Parser.Term.matchDiscr #[mkNullNode, discrExpr]

```

```

7802     elabMatchAux none #[discr] #[] mkNullNode expectedType
7803 | _ =>
7804     let stxNew ← `(let _discr := $discrExpr; nomatch _discr)
7805     withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
7806 | _ => throwUnsupportedSyntax
7807
7808 end Lean.Elab.Term
7809 ::::::::::::::
7810 Elab/MutualDef.lean
7811 ::::::::::::::
7812 /-
7813 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
7814 Released under Apache 2.0 license as described in the file LICENSE.
7815 Authors: Leonardo de Moura
7816 -/
7817 import Lean.Parser.Term
7818 import Lean.Meta.Closure
7819 import Lean.Meta.Check
7820 import Lean.Elab.Command
7821 import Lean.Elab.DefView
7822 import Lean.Elab.PreDefinition
7823 import Lean.Elab.DeclarationRange
7824
7825 namespace Lean.Elab
7826 open Lean.Parser.Term
7827
7828 /- DefView after elaborating the header. -/
7829 structure DefViewElabHeader where
7830   ref          : Syntax
7831   modifiers    : Modifiers
7832   kind         : DefKind
7833   shortDeclName : Name
7834   declName     : Name
7835   levelNames   : List Name
7836   numParams    : Nat
7837   type         : Expr -- including the parameters
7838   valueStx     : Syntax
7839   deriving Inhabited
7840
7841 namespace Term
7842
7843 open Meta
7844
7845 private def checkModifiers (m₁ m₂ : Modifiers) : TermElabM Unit := do
7846   unless m₁.isUnsafe == m₂.isUnsafe do
7847     throwError "cannot mix unsafe and safe definitions"
7848   unless m₁.isNoncomputable == m₂.isNoncomputable do

```

```

7849     throwError "cannot mix computable and non-computable definitions"
7850   unless m1.isPartial == m2.isPartial do
7851     throwError "cannot mix partial and non-partial definitions"
7852
7853 private def checkKinds (k1 k2 : DefKind) : TermElabM Unit := do
7854   unless k1.isExample == k2.isExample do
7855     throwError "cannot mix examples and definitions" -- Reason: we should discard examples
7856   unless k1.isTheorem == k2.isTheorem do
7857     throwError "cannot mix theorems and definitions" -- Reason: we will eventually elaborate theorems in `Task`s.
7858
7859 private def check (prevHeaders : Array DefViewElabHeader) (newHeader : DefViewElabHeader) : TermElabM Unit := do
7860   if newHeader.kind.isTheorem && newHeader.modifiers.isUnsafe then
7861     throwError "'unsafe' theorems are not allowed"
7862   if newHeader.kind.isTheorem && newHeader.modifiers.isPartial then
7863     throwError "'partial' theorems are not allowed, 'partial' is a code generation directive"
7864   if newHeader.kind.isTheorem && newHeader.modifiers.isNoncomputable then
7865     throwError "'theorem' subsumes 'noncomputable', code is not generated for theorems"
7866   if newHeader.modifiers.isNoncomputable && newHeader.modifiers.isUnsafe then
7867     throwError "'noncomputable unsafe' is not allowed"
7868   if newHeader.modifiers.isNoncomputable && newHeader.modifiers.isPartial then
7869     throwError "'noncomputable partial' is not allowed"
7870   if newHeader.modifiers.isPartial && newHeader.modifiers.isUnsafe then
7871     throwError "'unsafe' subsumes 'partial'"
7872   if h : 0 < prevHeaders.size then
7873     let firstHeader := prevHeaders.get (0, h)
7874     try
7875       unless newHeader.levelNames == firstHeader.levelNames do
7876         throwError "universe parameters mismatch"
7877       checkModifiers newHeader.modifiers firstHeader.modifiers
7878       checkKinds newHeader.kind firstHeader.kind
7879     catch
7880       | Exception.error ref msg => throw (Exception.error ref m!"invalid mutually recursive definitions, {msg}")
7881       | ex => throw ex
7882   else
7883     pure ()
7884
7885 private def registerFailedToInferDefTypeInfo (type : Expr) (ref : Syntax) : TermElabM Unit :=
7886   registerCustomErrorIfMVar type ref "failed to infer definition type"
7887
7888 private def elabHeaders (views : Array DefView) : TermElabM (Array DefViewElabHeader) := do
7889   let mut headers := #[]
7890   for view in views do
7891     let newHeader ← withRef view.ref do
7892       let (shortDeclName, declName, levelNames) ← expandDeclId (← getCurrNamespace) (← getLevelNames) view.declId view.modifiers
7893       addDeclarationRanges declName view.ref
7894       applyAttributesAt declName view.modifiers.attrs AttributeApplicationTime.beforeElaboration
7895       withDeclName declName <| withAutoBoundImplicit <| withLevelNames levelNames <|

```



```

7896   elabBinders view.binders.getArgs fun xs => do
7897     let refForElabFunType := view.value
7898     let type ← match view.type? with
7899       | some typeStx =>
7900         let type ← elabType typeStx
7901         registerFailedToInferDefTypeInfo type typeStx
7902         pure type
7903       | none =>
7904         let hole := mkHole refForElabFunType
7905         let type ← elabType hole
7906         registerFailedToInferDefTypeInfo type refForElabFunType
7907         pure type
7908     Term.synthesizeSyntheticMVarsNoPostponing
7909     let type ← mkForallFVars xs type
7910     let type ← mkForallFVars (← read).autoBoundImplicits.toArray type
7911     let type ← instantiateMVars type
7912     let xs ← addAutoBoundImplicits xs
7913     let levelNames ← getLevelNames
7914     if view.type?.isSome then
7915       let pendingMVarIds ← getMVars type
7916       discard <| logUnassignedUsingErrorInfos pendingMVarIds <|
7917         m!"\nwhen the resulting type of a declaration is explicitly provided, all holes (e.g., `\_`) in the header are resolved before"
7918     let newHeader := {
7919       ref      := view.ref,
7920       modifiers := view.modifiers,
7921       kind      := view.kind,
7922       shortDeclName := shortDeclName,
7923       declName   := declName,
7924       levelNames := levelNames,
7925       numParams  := xs.size,
7926       type       := type,
7927       valueStx   := view.value : DefViewElabHeader }
7928     check headers newHeader
7929     pure newHeader
7930   headers := headers.push newHeader
7931   pure headers
7932
7933 private partial def withFunLocalDecls {α} (headers : Array DefViewElabHeader) (k : Array Expr → TermElabM α) : TermElabM α :=
7934   let rec loop (i : Nat) (fvars : Array Expr) := do
7935     if h : i < headers.size then
7936       let header := headers.get {i, h}
7937       withLocalDecl header.shortDeclName BinderInfo.auxDecl header.type fun fvar => loop (i+1) (fvars.push fvar)
7938     else
7939       k fvars
7940   loop 0 #[]
7941
7942 private def expandWhereDeclsAsStructInst : Macro

```

```

7943 | `(whereDecls|where $[decls:letRecDecl$[;]?]*) => do
7944   let letIdDecls ← decls.mapM fun stx => match stx with
7945   | `(letRecDecl|$attrs:attributes $decl:letDecl) => Macro.throwErrorAt stx "attributes are 'where' elements are currently not suppo
7946   | `(letRecDecl|$decl:letPatDecl) => Macro.throwErrorAt stx "patterns are not allowed here"
7947   | `(letRecDecl|$decl:letEqnsDecl) => expandLetEqnsDecl decl
7948   | `(letRecDecl|$decl:letIdDecl) => pure decl
7949   | _ => Macro.throwUnsupported
7950   let structInstFields ← letIdDecls.mapM fun
7951   | stx@`(letIdDecl|$id:ident $[binders]* $[: $ty]? := $val) => withRef stx do
7952     let mut val := val
7953     if let some ty := ty? then
7954       val ← `(($val : $ty))
7955       val ← `(fun $[binders]* => $val:term)
7956       `(structInstField|$id:ident := $val)
7957     | _ => Macro.throwUnsupported
7958   `({ $[structInstFields,*] })
7959 | _ => Macro.throwUnsupported
7960
7961 /-
7962 Recall that
7963 ```
7964 def declValSimple      := leading_parser " :=\n" >> termParser >> optional Term.whereDecls
7965 def declValEqns       := leading_parser Term.matchAltsWhereDecls
7966 def declVal           := declValSimple <|> declValEqns <|> Term.whereDecls
7967 ```
7968 -/
7969 private def declValToTerm (declVal : Syntax) : MacroM Syntax := withRef declVal do
7970   if declVal.isOfKind `Lean.Parser.Command.declValSimple then
7971     expandWhereDeclsOpt declVal[2] declVal[1]
7972   else if declVal.isOfKind `Lean.Parser.Command.declValEqns then
7973     expandMatchAltsWhereDecls declVal[0]
7974   else if declVal.isOfKind `Lean.Parser.Term.whereDecls then
7975     expandWhereDeclsAsStructInst declVal
7976   else if declVal.isMissing then
7977     Macro.throwErrorAt declVal "declaration body is missing"
7978   else
7979     Macro.throwErrorAt declVal "unexpected declaration body"
7980
7981 private def elabFunValues (headers : Array DefViewElabHeader) : TermElabM (Array Expr) :=
7982   headers.mapM fun header => withDeclName header.declName $ withLevelNames header.levelNames do
7983     let valStx ← liftMacroM $ declValToTerm header.valueStx
7984     forallBoundedTelescope header.type header.numParams fun xs type => do
7985       let val ← elabTermEnsuringType valStx type
7986       mkLambdaFVars xs val
7987
7988 private def collectUsed (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
7989   : StateRefT CollectFVars.State MetaM Unit := do

```

```

7990 headers.forM fun header => collectUsedFVars header.type
7991 values.forM collectUsedFVars
7992 toLift.forM fun letRecToLift => do
7993   collectUsedFVars letRecToLift.type
7994   collectUsedFVars letRecToLift.val
7995
7996 private def removeUnusedVars (vars : Array Expr) (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
7997   : TermElabM (LocalContext × LocalInstances × Array Expr) := do
7998   let (_, used) ← (collectUsed headers values toLift).run {}
7999   removeUnused vars used
8000
8001 private def withUsed {α} (vars : Array Expr) (headers : Array DefViewElabHeader) (values : Array Expr) (toLift : List LetRecToLift)
8002   (k : Array Expr → TermElabM α) : TermElabM α := do
8003   let (lctx, localInsts, vars) ← removeUnusedVars vars headers values toLift
8004   withLCtx lctx localInsts $ k vars
8005
8006 private def isExample (views : Array DefView) : Bool :=
8007   views.any (·.kind.isExample)
8008
8009 private def isTheorem (views : Array DefView) : Bool :=
8010   views.any (·.kind.isTheorem)
8011
8012 private def instantiateMVarsAtHeader (header : DefViewElabHeader) : TermElabM DefViewElabHeader := do
8013   let type ← instantiateMVars header.type
8014   pure { header with type := type }
8015
8016 private def instantiateMVarsAtLetRecToLift (toLift : LetRecToLift) : TermElabM LetRecToLift := do
8017   let type ← instantiateMVars toLift.type
8018   let val ← instantiateMVars toLift.val
8019   pure { toLift with type := type, val := val }
8020
8021 private def typeHasRecFun (type : Expr) (funFVars : Array Expr) (letRecsToLift : List LetRecToLift) : Option FVarId :=
8022   let occ? := type.find? fun e => match e with
8023     | Expr.fvar fvarId _ => funFVars.contains e || letRecsToLift.any fun toLift => toLift.fvarId == fvarId
8024     | _ => false
8025   match occ? with
8026   | some (Expr.fvar fvarId _) => some fvarId
8027   | _ => none
8028
8029 private def getFunName (fvarId : FVarId) (letRecsToLift : List LetRecToLift) : TermElabM Name := do
8030   match (← findLocalDecl? fvarId) with
8031   | some decl => pure decl.userName
8032   | none =>
8033     /- Recall that the FVarId of nested let-recs are not in the current local context. -/
8034     match letRecsToLift.findSome? fun toLift => if toLift.fvarId == fvarId then some toLift.shortDeclName else none with
8035     | none => throwError "unknown function"
8036     | some n => pure n

```

```

8037
8038 /-
8039 Ensures that the of let-rec definition types do not contain functions being defined.
8040 In principle, this test can be improved. We could perform it after we separate the set of functions is strongly connected components.
8041 However, this extra complication doesn't seem worth it.
8042 -/
8043 private def checkLetRecsToLiftTypes (funVars : Array Expr) (letRecsToLift : List LetRecToLift) : TermElabM Unit :=
8044   letRecsToLift.forM fun toLift =>
8045     match typeHasRecFun toLift.type funVars letRecsToLift with
8046     | none      => pure ()
8047     | some fvarId => do
8048       let fnName ← getFunName fvarId letRecsToLift
8049       throwErrorAt toLift.ref "invalid type in 'let rec', it uses '{fnName}' which is being defined simultaneously"
8050
8051 namespace MutualClosure
8052
8053 /- A mapping from FVarId to Set of FVarIds. -/
8054 abbrev UsedFVarsMap := NameMap NameSet
8055
8056 /-
8057 Create the `UsedFVarsMap` mapping that takes the variable id for the mutually recursive functions being defined to the set of
8058 free variables in its definition.
8059
8060 For `mainFVars`, this is just the set of section variables `sectionVars` used.
8061 For nested let-rec functions, we collect their free variables.
8062
8063 Recall that a `let rec` expressions are encoded as follows in the elaborator.
8064 ```lean
8065 let rec
8066   f : A := t,
8067   g : B := s;
8068 body
8069 ```
8070 is encoded as
8071 ```lean
8072 let f : A := ?m₁;
8073 let g : B := ?m₂;
8074 body
8075 ```
8076 where `?m₁` and `?m₂` are synthetic opaque metavariables. That are assigned by this module.
8077 We may have nested `let rec`s.
8078 ```lean
8079 let rec f : A :=
8080   let rec g : B := t;
8081   s;
8082 body
8083 ```

```

```

8084 is encoded as
8085 ```lean
8086 let f : A := ?m1;
8087 body
8088 ```
8089 and the body of `f` is stored the field `val` of a `LetRecToLift`. For the example above,
8090 we would have a `LetRecToLift` containing:
8091 ```
8092 {
8093   mvarId := m1,
8094   val    := `(let g : B := ?m2; body)
8095   ...
8096 }
8097 ```
8098 Note that `g` is not a free variable at `(let g : B := ?m2; body)`. We recover the fact that
8099 `f` depends on `g` because it contains `m2`
8100 -/
8101 private def mkInitialUsedFVarsMap (mctx : MetavarContext) (sectionVars : Array Expr) (mainFVarIds : Array FVarId) (letRecsToLift : List (LetRecToLift)) : UsedFVarsMap := do
8102   : UsedFVarsMap := do
8103     let mut sectionVarSet := {}
8104     for var in sectionVars do
8105       sectionVarSet := sectionVarSet.insert var.fvarId!
8106     let mut usedFVarMap := {}
8107     for mainFVarId in mainFVarIds do
8108       usedFVarMap := usedFVarMap.insert mainFVarId sectionVarSet
8109     for toLift in letRecsToLift do
8110       let state := Lean.collectFVars {} toLift.val
8111       let state := Lean.collectFVars state toLift.type
8112       let mut set := state.fvarSet
8113       /- toLift.val may contain metavariables that are placeholders for nested let-recs. We should collect the fvarId
8114          for the associated let-rec because we need this information to compute the fixpoint later. -/
8115       let mvarIds := (toLift.val.collectMVars {}).result
8116       for mvarId in mvarIds do
8117         match letRecsToLift.findSome? fun (toLift : LetRecToLift) => if toLift.mvarId == mctx.getDelayedRoot mvarId then some toLift.fvarId
8118         | some fvarId => set := set.insert fvarId
8119         | none        => pure ()
8120       usedFVarMap := usedFVarMap.insert toLift.fvarId set
8121     pure usedFVarMap
8122
8123 /-
8124 The let-recs may invoke each other. Example:
8125 ```
8126 let rec
8127   f (x : Nat) := g x + y
8128   g : Nat → Nat
8129   | 0    => 1
8130   | x+1 => f x + z

```

```

8131 ``
8132 `y` is free variable in `f`, and `z` is a free variable in `g`.
8133 To close `f` and `g`, `y` and `z` must be in the closure of both.
8134 That is, we need to generate the top-level definitions.
8135 ``
8136 def f (y z x : Nat) := g y z x + y
8137 def g (y z : Nat) : Nat → Nat
8138   | 0 => 1
8139   | x+1 => f y z x + z
8140 ``
8141 -/
8142 namespace FixPoint
8143
8144 structure State where
8145   usedFVarsMap : UsedFVarsMap := {}
8146   modified      : Bool         := false
8147
8148 abbrev M := ReaderT (List FVarId) $ StateM State
8149
8150 private def isModified : M Bool := do pure (← get).modified
8151 private def resetModified : M Unit := modify fun s => { s with modified := false }
8152 private def markModified : M Unit := modify fun s => { s with modified := true }
8153 private def getUsedFVarsMap : M UsedFVarsMap := do pure (← get).usedFVarsMap
8154 private def modifyUsedFVars (f : UsedFVarsMap → UsedFVarsMap) : M Unit := modify fun s => { s with usedFVarsMap := f s.usedFVarsMap }
8155
8156 -- merge s2 into s1
8157 private def merge (s1 s2 : NameSet) : M NameSet :=
8158   s2.foldM (init := s1) fun s1 k => do
8159     if s1.contains k then
8160       pure s1
8161     else
8162       markModified
8163       pure $ s1.insert k
8164
8165 private def updateUsedVarsOf (fvarId : FVarId) : M Unit := do
8166   let usedFVarsMap ← getUsedFVarsMap
8167   match usedFVarsMap.find? fvarId with
8168   | none      => pure ()
8169   | some fvarIds =>
8170     let fvarIdsNew ← fvarIds.foldM (init := fvarIds) fun fvarIdsNew fvarId' =>
8171       if fvarId == fvarId' then
8172         pure fvarIdsNew
8173     else
8174       match usedFVarsMap.find? fvarId' with
8175       | none => pure fvarIdsNew
8176       /- We are being sloppy here `otherFVarIds` may contain free variables that are
8177         not in the context of the let-rec associated with fvarId.

```

```

8178         We filter these out-of-context free variables later. -/
8179         | some otherFVarIds => merge fvarIdsNew otherFVarIds
8180     modifyUsedFVars fun usedFVars => usedFVars.insert fvarId fvarIdsNew
8181
8182 private partial def fixpoint : Unit → M Unit
8183 | _ => do
8184     resetModified
8185     let letRecFVarIds ← read
8186     letRecFVarIds.forM updateUsedVarsOf
8187     if (← isModified) then
8188         fixpoint ()
8189
8190 def run (letRecFVarIds : List FVarId) (usedFVarsMap : UsedFVarsMap) : UsedFVarsMap :=
8191     let (_, s) := ((fixpoint ()).run letRecFVarIds).run { usedFVarsMap := usedFVarsMap }
8192     s.usedFVarsMap
8193
8194 end FixPoint
8195
8196 abbrev FreeVarMap := NameMap (Array FVarId)
8197
8198 private def mkFreeVarMap
8199     (mctx : MetavarContext) (sectionVars : Array Expr) (mainFVarIds : Array FVarId)
8200     (recFVarIds : Array FVarId) (letRecsToLift : List LetRecToLift) : FreeVarMap := do
8201     let usedFVarsMap := mkInitialUsedFVarsMap mctx sectionVars mainFVarIds letRecsToLift
8202     let letRecFVarIds := letRecsToLift.map fun toLift => toLift.fvarId
8203     let usedFVarsMap := FixPoint.run letRecFVarIds usedFVarsMap
8204     let mut freeVarMap := {}
8205     for toLift in letRecsToLift do
8206         let lctx := toLift.lctx
8207         let fvarIdsSet := (usedFVarsMap.find? toLift.fvarId).get!
8208         let fvarIds := fvarIdsSet.fold (init := #[]) fun fvarIds fvarId =>
8209             if lctx.contains fvarId && !recFVarIds.contains fvarId then
8210                 fvarIds.push fvarId
8211             else
8212                 fvarIds
8213     freeVarMap := freeVarMap.insert toLift.fvarId fvarIds
8214     pure freeVarMap
8215
8216 structure ClosureState where
8217     newLocalDecls : Array LocalDecl := #[]
8218     localDecls : Array LocalDecl := #[]
8219     newLetDecls : Array LocalDecl := #[]
8220     exprArgs : Array Expr := #[]
8221
8222 private def pickMaxFVar? (lctx : LocalContext) (fvarIds : Array FVarId) : Option FVarId :=
8223     fvarIds.getMax? fun fvarId1 fvarId2 => (lctx.get! fvarId1).index < (lctx.get! fvarId2).index
8224

```

```

8225 private def preprocess (e : Expr) : TermElabM Expr := do
8226   let e ← instantiateMVars e
8227   -- which let-decls are dependent. We say a let-decl is dependent if its lambda abstraction is type incorrect.
8228   Meta.check e
8229   pure e
8230
8231 /- Push free variables in `s` to `toProcess` if they are not already there. -/
8232 private def pushNewVars (toProcess : Array FVarId) (s : CollectFVars.State) : Array FVarId :=
8233   s.fvarSet.fold (init := toProcess) fun toProcess fvarId =>
8234     if toProcess.contains fvarId then toProcess else toProcess.push fvarId
8235
8236 private def pushLocalDecl (toProcess : Array FVarId) (fvarId : FVarId) (userName : Name) (type : Expr) (bi := BinderInfo.default)
8237   : StateRefT ClosureState TermElabM (Array FVarId) := do
8238   let type ← preprocess type
8239   modify fun s => { s with
8240     newLocalDecls := s.newLocalDecls.push $ LocalDecl.cdecl arbitrary fvarId userName type bi,
8241     exprArgs      := s.exprArgs.push (mkFVar fvarId)
8242   }
8243   pure $ pushNewVars toProcess (collectFVars {} type)
8244
8245 private partial def mkClosureForAux (toProcess : Array FVarId) : StateRefT ClosureState TermElabM Unit := do
8246   let lctx ← getLCtx
8247   match pickMaxFVar? lctx toProcess with
8248   | none      => pure ()
8249   | some fvarId =>
8250     trace[Elab.definition.mkClosure] "toProcess: {toProcess.map mkFVar}, maxVar: {mkFVar fvarId}"
8251     let toProcess := toProcess.erase fvarId
8252     let localDecl ← getLocalDecl fvarId
8253     match localDecl with
8254     | LocalDecl.cdecl _ _ userName type bi =>
8255       let toProcess ← pushLocalDecl toProcess fvarId userName type bi
8256       mkClosureForAux toProcess
8257     | LocalDecl.ldecl _ _ userName type val _ =>
8258       let zetaFVarIds ← getZetaFVarIds
8259       if !zetaFVarIds.contains fvarId then
8260         /- Non-dependent let-decl. See comment at src/Lean/Meta/Closure.lean -/
8261         let toProcess ← pushLocalDecl toProcess fvarId userName type
8262         mkClosureForAux toProcess
8263       else
8264         /- Dependent let-decl. -/
8265         let type ← preprocess type
8266         let val  ← preprocess val
8267         modify fun s => { s with
8268           newLetDecls := s.newLetDecls.push $ LocalDecl.ldecl arbitrary fvarId userName type val false,
8269           /- We don't want to interleave let and lambda declarations in our closure. So, we expand any occurrences of fvarId
8270             at `newLocalDecls` and `localDecls` -/
8271           newLocalDecls := s.newLocalDecls.map (replaceFVarIdAtLocalDecl fvarId val),

```



```

8272         localDecls := s.localDecls.map (replaceFVarIdAtLocalDecl fvarId val)
8273     }
8274     mkClosureForAux (pushNewVars toProcess (collectFVars (collectFVars {}) type) val))
8275
8276 private partial def mkClosureFor (freeVars : Array FVarId) (localDecls : Array LocalDecl) : TermElabM ClosureState := do
8277     let (_, s) ← (mkClosureForAux freeVars).run { localDecls := localDecls }
8278     pure { s with
8279         newLocalDecls := s.newLocalDecls.reverse,
8280         newLetDecls   := s.newLetDecls.reverse,
8281         exprArgs      := s.exprArgs.reverse
8282     }
8283
8284 structure LetRecClosure where
8285     ref      : Syntax
8286     localDecls : Array LocalDecl
8287     closed   : Expr -- expression used to replace occurrences of the let-rec FVarId
8288     toLift   : LetRecToLift
8289
8290 private def mkLetRecClosureFor (toLift : LetRecToLift) (freeVars : Array FVarId) : TermElabM LetRecClosure := do
8291     let lctx := toLift.lctx
8292     withLCtx lctx toLift.localInstances do
8293         lambdaTelescope toLift.val fun xs val => do
8294             let type ← instantiateForall toLift.type xs
8295             let lctx ← getLCtx
8296             let s ← mkClosureFor freeVars $ xs.map fun x => lctx.get! x.fvarId!
8297             let type := Closure.mkForall s.localDecls $ Closure.mkForall s.newLetDecls type
8298             let val  := Closure.mkLambda s.localDecls $ Closure.mkLambda s.newLetDecls val
8299             let c    := mkAppN (Lean.mkConst toLift.declName) s.exprArgs
8300             assignExprMVar toLift.mvarId c
8301             return {
8302                 ref      := toLift.ref
8303                 localDecls := s.newLocalDecls
8304                 closed   := c
8305                 toLift   := { toLift with val := val, type := type }
8306             }
8307
8308 private def mkLetRecClosures (letRecsToLift : List LetRecToLift) (freeVarMap : FreeVarMap) : TermElabM (List LetRecClosure) :=
8309     letRecsToLift.mapM fun toLift => mkLetRecClosureFor toLift (freeVarMap.find? toLift.fvarId).get!
8310
8311 /- Mapping from FVarId of mutually recursive functions being defined to "closure" expression. -/
8312 abbrev Replacement := NameMap Expr
8313
8314 def insertReplacementForMainFns (r : Replacement) (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainFVars : Array |
8315     mainFVars.size.fold (init := r) fun i r =>
8316         r.insert mainFVars[i].fvarId! (mkAppN (Lean.mkConst mainHeaders[i].declName) sectionVars)
8317
8318

```

```

8319 def insertReplacementForLetRecs (r : Replacement) (letRecClosures : List LetRecClosure) : Replacement :=
8320   letRecClosures.foldl (init := r) fun r c =>
8321     r.insert c.toLift.fvarId c.closed
8322
8323 def Replacement.apply (r : Replacement) (e : Expr) : Expr :=
8324   e.replace fun e => match e with
8325   | Expr.fvar fvarId _ => match r.find? fvarId with
8326   | some c => some c
8327   | _      => none
8328   | _      => none
8329
8330 def pushMain (preDefs : Array PreDefinition) (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainVals : Array Expr)
8331   : TermElabM (Array PreDefinition) :=
8332   mainHeaders.size.foldM (init := preDefs) fun i preDefs => do
8333     let header := mainHeaders[i]
8334     let val ← mkLambdaFVars sectionVars mainVals[i]
8335     let type ← mkForallFVars sectionVars header.type
8336     return preDefs.push {
8337       ref      := getDeclarationSelectionRef header.ref
8338       kind     := header.kind
8339       declName := header.declName
8340       levelParams := [], -- we set it later
8341       modifiers := header.modifiers
8342       type     := type
8343       value    := val
8344     }
8345
8346 def pushLetRecs (preDefs : Array PreDefinition) (letRecClosures : List LetRecClosure) (kind : DefKind) (modifiers : Modifiers) : Array P
8347   letRecClosures.foldl (init := preDefs) fun preDefs c =>
8348     let type := Closure.mkForall c.localDecls c.toLift.type
8349     let val  := Closure.mkLambda c.localDecls c.toLift.val
8350     preDefs.push {
8351       ref      := c.ref
8352       kind     := kind
8353       declName := c.toLift.declName
8354       levelParams := [] -- we set it later
8355       modifiers := { modifiers with attrs := c.toLift.attrs }
8356       type     := type
8357       value    := val
8358     }
8359
8360 def getKindForLetRecs (mainHeaders : Array DefViewElabHeader) : DefKind :=
8361   if mainHeaders.any fun h => h.kind.isTheorem then DefKind.«theorem»
8362   else DefKind.«def»
8363
8364 def getModifiersForLetRecs (mainHeaders : Array DefViewElabHeader) : Modifiers := {
8365   isNoncomputable := mainHeaders.any fun h => h.modifiers.isNoncomputable,

```

```

8366 isPartial      := mainHeaders.any fun h => h.modifiers.isPartial,
8367 isUnsafe       := mainHeaders.any fun h => h.modifiers.isUnsafe
8368 }
8369
8370 /-
8371 - `sectionVars`: The section variables used in the `mutual` block.
8372 - `mainHeaders`: The elaborated header of the top-level definitions being defined by the mutual block.
8373 - `mainFVars`: The auxiliary variables used to represent the top-level definitions being defined by the mutual block.
8374 - `mainVals`: The elaborated value for the top-level definitions
8375 - `letRecsToLift`: The let-rec's definitions that need to be lifted
8376 -/
8377 def main (sectionVars : Array Expr) (mainHeaders : Array DefViewElabHeader) (mainFVars : Array Expr) (mainVals : Array Expr) (letRecsToLift : TermElabM (Array PreDefinition) := do
8378   : TermElabM (Array PreDefinition) := do
8379   -- Store in recFVarIds the fvarId of every function being defined by the mutual block.
8380   let mainFVarIds := mainFVars.map Expr.fvarId!
8381   let recFVarIds := (letRecsToLift.toArray.map fun toLift => toLift.fvarId) ++ mainFVarIds
8382   -- Compute the set of free variables (excluding `recFVarIds`) for each let-rec.
8383   let mctx ← getMCtx
8384   let freeVarMap := mkFreeVarMap mctx sectionVars mainFVarIds recFVarIds letRecsToLift
8385   resetZetaFVarIds
8386   withTrackingZeta do
8387     -- By checking `toLift.type` and `toLift.val` we populate `zetaFVarIds`. See comments at `src/Lean/Meta/Closure.lean`.
8388     letRecsToLift.forM fun toLift => withLCtx toLift.lctx toLift.localInstances do Meta.check toLift.type; Meta.check toLift.val
8389     let letRecClosures ← mkLetRecClosures letRecsToLift freeVarMap
8390     -- mkLetRecClosures assign metavariables that were placeholders for the lifted declarations.
8391     let mainVals ← mainVals.mapM (instantiateMVars ·)
8392     let mainHeaders ← mainHeaders.mapM instantiateMVarsAtHeader
8393     let letRecClosures ← letRecClosures.mapM fun closure => do pure { closure with toLift := (← instantiateMVarsAtLetRecToLift closure.toLift) }
8394     -- Replace fvarIds for functions being defined with closed terms
8395     let r := insertReplacementForMainFns {} sectionVars mainHeaders mainFVars
8396     let r := insertReplacementForLetRecs r letRecClosures
8397     let mainVals := mainVals.map r.apply
8398     let mainHeaders := mainHeaders.map fun h => { h with type := r.apply h.type }
8399     let letRecClosures := letRecClosures.map fun c => { c with toLift := { c.toLift with type := r.apply c.toLift.type, val := r.apply c.toLift.val } }
8400     let letRecKind := getKindForLetRecs mainHeaders
8401     let letRecMods := getModifiersForLetRecs mainHeaders
8402     pushMain (pushLetRecs #[] letRecClosures letRecKind letRecMods) sectionVars mainHeaders mainVals
8403
8404 end MutualClosure
8405
8406 private def getAllUserLevelNames (headers : Array DefViewElabHeader) : List Name :=
8407   if h : 0 < headers.size then
8408     -- Recall that all top-level functions must have the same levels. See `check` method above
8409     (headers.get (0, h)).levelNames
8410   else
8411     []
8412

```

```

8413 /- Eagerly convert universe metavariables occurring in theorem headers to universe parameters. -/
8414 private def levelMVarToParamHeaders (views : Array DefView) (headers : Array DefViewElabHeader) : TermElabM (Array DefViewElabHeader) :=
8415   let rec process : StateRefT Nat TermElabM (Array DefViewElabHeader) := do
8416     let mut newHeaders := #[]
8417     for view in views, header in headers do
8418       if view.kind.isTheorem then
8419         newHeaders := newHeaders.push { header with type := (← levelMVarToParam' header.type) }
8420       else
8421         newHeaders := newHeaders.push header
8422     return newHeaders
8423   let newHeaders ← process.run' 1
8424   newHeaders.mapM fun header => return { header with type := (← instantiateMVars header.type) }
8425
8426 def elabMutualDef (vars : Array Expr) (views : Array DefView) : TermElabM Unit := do
8427   let scopeLevelNames ← getLevelNames
8428   let headers ← elabHeaders views
8429   let headers ← levelMVarToParamHeaders views headers
8430   let allUserLevelNames := getAllUserLevelNames headers
8431   withFunLocalDecls headers fun funFVars => do
8432     let values ← elabFunValues headers
8433     Term.synthesizeSyntheticMVarsNoPostponing
8434     let values ← values.mapM (instantiateMVars ·)
8435     let headers ← headers.mapM instantiateMVarsAtHeader
8436     let letRecsToLift ← getLetRecsToLift
8437     let letRecsToLift ← letRecsToLift.mapM instantiateMVarsAtLetRecToLift
8438     checkLetRecsToLiftTypes funFVars letRecsToLift
8439     withUsed vars headers values letRecsToLift fun vars => do
8440       let preDefs ← MutualClosure.main vars headers funFVars values letRecsToLift
8441       let preDefs ← levelMVarToParamPreDecls preDefs
8442       let preDefs ← instantiateMVarsAtPreDecls preDefs
8443       let preDefs ← fixLevelParams preDefs scopeLevelNames allUserLevelNames
8444       if isExample views then
8445         withoutModifyingEnv <| addPreDefinitions preDefs
8446       else
8447         addPreDefinitions preDefs
8448
8449 end Term
8450 namespace Command
8451
8452 def elabMutualDef (ds : Array Syntax) : CommandElabM Unit := do
8453   let views ← ds.mapM fun d => do
8454     let modifiers ← elabModifiers d[0]
8455     mkDefView modifiers d[1]
8456   runTermElabM none fun vars => Term.elabMutualDef vars views
8457
8458 end Command
8459 end Lean.Elab

```

```

8460 ::::::::::::::
8461 Elab/Open.lean
8462 ::::::::::::::
8463 /-
8464 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
8465 Released under Apache 2.0 license as described in the file LICENSE.
8466 Authors: Leonardo de Moura
8467 -/
8468 import Lean.Elab.Log
8469
8470 namespace Lean.Elab
8471 namespace OpenDecl
8472
8473 variable [Monad m] [STWorld IO.RealWorld m] [MonadEnv m]
8474 variable [MonadExceptOf Exception m] [MonadRef m] [AddErrorMessageContext m]
8475 variable [AddMessageContext m] [MonadLiftT (ST IO.RealWorld) m] [MonadLog m]
8476
8477 structure State where
8478   openDecls      : List OpenDecl
8479   currNamespace  : Name
8480
8481 abbrev M := StateRefT State m
8482
8483 instance : MonadResolveName (M (m := m)) where
8484   getCurrNamespace := return (← get).currNamespace
8485   getOpenDecls     := return (← get).openDecls
8486
8487 private def addOpenDecl (decl : OpenDecl) : M (m:=m) Unit :=
8488   modify fun s => { s with openDecls := decl :: s.openDecls }
8489
8490 private def elabOpenSimple (n : Syntax) : M (m:=m) Unit :=
8491   -- `open` id+
8492   for ns in n[0].getArgs do
8493     let ns ← resolveNamespace ns.getId
8494     addOpenDecl (OpenDecl.simple ns [])
8495     activateScoped ns
8496
8497 -- `open` id `(` id+ `)`
8498 private def elabOpenOnly (n : Syntax) : M (m:=m) Unit := do
8499   let ns ← resolveNamespace n[0].getId
8500   for idStx in n[2].getArgs do
8501     let id := idStx.getId
8502     let declName := ns ++ id
8503     if (← getEnv).contains declName then
8504       addOpenDecl (OpenDecl.explicit id declName)
8505     else
8506       withRef idStx <| logUnknownDecl declName

```

```

8507
8508 -- `open` id `hiding` id+
8509 private def elabOpenHiding (n : Syntax) : M (m:=m) Unit := do
8510   let ns ← resolveNamespace n[0].getId
8511   let mut ids : List Name := []
8512   for idStx in n[2].getArgs do
8513     let id := idStx.getId
8514     let declName := ns ++ id
8515     if (← getEnv).contains declName then
8516       ids := id::ids
8517     else
8518       withRef idStx <| logUnknownDecl declName
8519   addOpenDecl (OpenDecl.simple ns ids)
8520
8521 -- `open` id `renaming` sepBy (id `->` id) `,`
8522 private def elabOpenRenaming (n : Syntax) : M (m:=m) Unit := do
8523   let ns ← resolveNamespace n[0].getId
8524   for stx in n[2].getSepArgs do
8525     let fromId := stx[0].getId
8526     let toId := stx[2].getId
8527     let declName := ns ++ fromId
8528     if (← getEnv).contains declName then
8529       addOpenDecl (OpenDecl.explicit toId declName)
8530     else
8531       withRef stx do logUnknownDecl declName
8532
8533 def elabOpenDecl [MonadResolveName m] (openDeclStx : Syntax) : m (List OpenDecl) := do
8534   StateRefT'.run' (s := { openDecls := (← getOpenDecls), currNamespace := (← getCurrNamespace) }) do
8535     if openDeclStx.getKind == ``Parser.Command.openSimple then
8536       elabOpenSimple openDeclStx
8537     else if openDeclStx.getKind == ``Parser.Command.openOnly then
8538       elabOpenOnly openDeclStx
8539     else if openDeclStx.getKind == ``Parser.Command.openHiding then
8540       elabOpenHiding openDeclStx
8541     else
8542       elabOpenRenaming openDeclStx
8543   return (← get).openDecls
8544
8545 end OpenDecl
8546
8547 export OpenDecl (elabOpenDecl)
8548
8549 end Lean.Elab::::::::::::::::::
8550 Elab/PreDefinition.lean
8551 ::::::::::::::::::::
8552 /-
8553 Copyright (c) 2020 Microsoft Corporation. All rights reserved.

```

8554 Released under Apache 2.0 license as described in the file LICENSE.

8555 Authors: Leonardo de Moura

8556 -/

8557 import Lean.Elab.PreDefinition.Basic

8558 import Lean.Elab.PreDefinition.Structural

8559 import Lean.Elab.PreDefinition.Main

8560 import Lean.Elab.PreDefinition.MkInhabitant

8561 ::::::::::::::

8562 Elab/Print.lean

8563 ::::::::::::::

8564 /-

8565 Copyright (c) 2020 Microsoft Corporation. All rights reserved.

8566 Released under Apache 2.0 license as described in the file LICENSE.

8567 Authors: Leonardo de Moura

8568 -/

8569 import Lean.Util.FoldConsts

8570 import Lean.Elab.Command

8571

8572 namespace Lean.Elab.Command

8573

8574 private def throwUnknownId (id : Name) : CommandElabM Unit :=

8575 throwError "unknown identifier '{mkConst id}'"

8576

8577 private def levelParamsToMessageData (levelParams : List Name) : MessageData :=

8578 match levelParams with

8579 | [] => ""

8580 | u::us => do

8581 let mut m := m! ".\{{{u}"

8582 for u in us do

8583 m := m ++ ", " ++ u

8584 return m ++ "}"

8585

8586 private def mkHeader (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (safety : DefinitionSafety) : CommandElabM MessageData :=

8587 let m : MessageData :=

8588 match safety with

8589 | DefinitionSafety.unsafe => "unsafe "

8590 | DefinitionSafety.partial => "partial "

8591 | DefinitionSafety.safe => ""

8592 let m := if isProtected (← getEnv) id then m ++ "protected " else m

8593 let (m, id) := match privateToUserName? id with

8594 | some id => (m ++ "private ", id)

8595 | none => (m, id)

8596 let m := m ++ kind ++ " " ++ id ++ levelParamsToMessageData levelParams ++ " : " ++ type

8597 pure m

8598

8599 private def mkHeader' (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (isUnsafe : Bool) : CommandElabM MessageData :=

8600 mkHeader kind id levelParams type (if isUnsafe then DefinitionSafety.unsafe else DefinitionSafety.safe)

```

8601
8602 private def printDefLike (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (value : Expr) (safety := DefinitionSafety.
8603   let m ← mkHeader kind id levelParams type safety
8604   let m := m ++ " :=" ++ Format.line ++ value
8605   logInfo m
8606
8607 private def printAxiomLike (kind : String) (id : Name) (levelParams : List Name) (type : Expr) (isUnsafe := false) : CommandElabM Unit :=
8608   logInfo (← mkHeader' kind id levelParams type isUnsafe)
8609
8610 private def printQuot (kind : QuotKind) (id : Name) (levelParams : List Name) (type : Expr) : CommandElabM Unit := do
8611   printAxiomLike "Quotient primitive" id levelParams type
8612
8613 private def printInduct (id : Name) (levelParams : List Name) (numParams : Nat) (numIndices : Nat) (type : Expr)
8614   (ctors : List Name) (isUnsafe : Bool) : CommandElabM Unit := do
8615   let mut m ← mkHeader' "inductive" id levelParams type isUnsafe
8616   m := m ++ Format.line ++ "constructors:"
8617   for ctor in ctors do
8618     let cinfo ← getConstInfo ctor
8619     m := m ++ Format.line ++ ctor ++ " : " ++ cinfo.type
8620   logInfo m
8621
8622 private def printIdCore (id : Name) : CommandElabM Unit := do
8623   match (← getEnv).find? id with
8624   | ConstantInfo.axiomInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "axiom" id us t u
8625   | ConstantInfo.defnInfo { levelParams := us, type := t, value := v, safety := s, .. } => printDefLike "def" id us t v s
8626   | ConstantInfo.thmInfo { levelParams := us, type := t, value := v, .. } => printDefLike "theorem" id us t v
8627   | ConstantInfo.opaqueInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "constant" id us t u
8628   | ConstantInfo.quotInfo { kind := kind, levelParams := us, type := t, .. } => printQuot kind id us t
8629   | ConstantInfo.ctorInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "constructor" id us t u
8630   | ConstantInfo.recInfo { levelParams := us, type := t, isUnsafe := u, .. } => printAxiomLike "recursor" id us t u
8631   | ConstantInfo.inductInfo { levelParams := us, numParams := numParams, numIndices := numIndices, type := t, ctors := ctors, isUnsafe :=
8632     printInduct id us numParams numIndices t ctors u
8633   | none => throwUnknownId id
8634
8635 private def printId (id : Syntax) : CommandElabM Unit := do
8636   let cs ← resolveGlobalConstWithInfos id
8637   cs.forM printIdCore
8638
8639 @[builtinCommandElab «print»] def elabPrint : CommandElab
8640 | `(#print$tk $id:ident) => withRef tk <| printId id
8641 | `(#print$tk $s:strLit) => logInfoAt tk s.isStrLit?.get!
8642 | _ => throwError "invalid #print command"
8643
8644 namespace CollectAxioms
8645
8646 structure State where
8647   visited : NameSet := {}

```



```

8648   axioms   : Array Name := #[]
8649
8650 abbrev M := ReaderT Environment $ StateM State
8651
8652 partial def collect (c : Name) : M Unit := do
8653   let collectExpr (e : Expr) : M Unit := e.getUsedConstants.forM collect
8654   let s ← get
8655   unless s.visited.contains c do
8656     modify fun s => { s with visited := s.visited.insert c }
8657     let env ← read
8658     match env.find? c with
8659     | some (ConstantInfo.axiomInfo _) => modify fun s => { s with axioms := s.axioms.push c }
8660     | some (ConstantInfo.defnInfo v)  => collectExpr v.type *> collectExpr v.value
8661     | some (ConstantInfo.thmInfo v)   => collectExpr v.type *> collectExpr v.value
8662     | some (ConstantInfo.opaqueInfo v) => collectExpr v.type *> collectExpr v.value
8663     | some (ConstantInfo.quotInfo _)  => pure ()
8664     | some (ConstantInfo.ctorInfo v)   => collectExpr v.type
8665     | some (ConstantInfo.recInfo v)    => collectExpr v.type
8666     | some (ConstantInfo.inductInfo v) => collectExpr v.type *> v.ctors.forM collect
8667     | none                             => pure ()
8668
8669 end CollectAxioms
8670
8671 private def printAxiomsOf (constName : Name) : CommandElabM Unit := do
8672   let env ← getEnv
8673   let (_, s) := ((CollectAxioms.collect constName).run env).run {}
8674   if s.axioms.isEmpty then
8675     logInfo m! "{constName}' does not depend on any axioms"
8676   else
8677     logInfo m! "{constName}' depends on axioms: {s.axioms.toList}"
8678
8679 @[builtinCommandElab «printAxioms»] def elabPrintAxioms : CommandElab
8680 | `(#print%$tk axioms $id) => withRef tk do
8681   let cs ← resolveGlobalConstWithInfos id
8682   cs.forM printAxiomsOf
8683 | _ => throwUnsupportedSyntax
8684
8685 end Lean.Elab.Command
8686 ::::::::::::::
8687 Elab/Quotation.lean
8688 ::::::::::::::
8689 /-
8690 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
8691 Released under Apache 2.0 license as described in the file LICENSE.
8692 Authors: Sebastian Ullrich
8693
8694 Elaboration of syntax quotations as terms and patterns (in `match_syntax`). See also `./Hygiene.lean` for the basic

```

```

8695 hygiene workings and data types.
8696 -/
8697 import Lean.Syntax
8698 import Lean.ResolveName
8699 import Lean.Elab.Term
8700 import Lean.Elab.Quotation.Util
8701 import Lean.Parser.Term
8702
8703 namespace Lean.Elab.Term.Quotation
8704 open Lean.Parser.Term
8705 open Lean.Syntax
8706 open Meta
8707
8708 register_builtin_option hygiene : Bool := {
8709   defValue := true
8710   descr   := "Annotate identifiers in quotations such that they are resolved relative to the scope at their declaration, not that at th
8711 }
8712
8713 /- `C[$(e)]` ~> `let a := e; C[$a]`. Used in the implementation of antiquot splices. -/
8714 private partial def floatOutAntiquotTerms : Syntax → StateT (Syntax → TermElabM Syntax) TermElabM Syntax
8715 | stx@(Syntax.node k args) => do
8716   if isAntiquot stx && !isEscapedAntiquot stx then
8717     let e := getAntiquotTerm stx
8718     if !e.isIdent || !e.getId.isAtomic then
8719       return ← withFreshMacroScope do
8720         let a ← `(a)
8721         modify (fun cont stx => `(let $a:ident := $e; $stx) : TermElabM _)
8722         stx.setArg 2 a
8723     Syntax.node k (← args.mapM floatOutAntiquotTerms)
8724 | stx => pure stx
8725
8726 private def getSepFromSplice (splice : Syntax) : Syntax := do
8727   let Syntax.atom _ sep ← getAntiquotSpliceSuffix splice | unreachable!
8728   Syntax.mkStrLit (sep.dropRight 1)
8729
8730 partial def mkTuple : Array Syntax → TermElabM Syntax
8731 | #[] => `(Unit.unit)
8732 | #[e] => e
8733 | es => do
8734   let stx ← mkTuple (es.eraseIdx 0)
8735   `(Prod.mk $(es[0]) $stx)
8736
8737 def resolveSectionVariable (sectionVars : NameMap Name) (id : Name) : List (Name × List String) :=
8738   -- decode macro scopes from name before recursion
8739   let extractionResult := extractMacroScopes id
8740   let rec loop : Name → List String → List (Name × List String)
8741     | id@(Name.str p s _), projs =>

```

```

8742     -- NOTE: we assume that macro scopes always belong to the projected constant, not the projections
8743     let id := { extractionResult with name := id }.review
8744     match sectionVars.find? id with
8745     | some newId => [(newId, projs)]
8746     | none       => loop p (s::projs)
8747   | _, _ => []
8748   loop extractionResult.name []
8749
8750 -- Elaborate the content of a syntax quotation term
8751 private partial def quoteSyntax : Syntax → TermElabM Syntax
8752 | Syntax.ident info rawVal val preresolved => do
8753   if !hygiene.get (← getOptions) then
8754     return ← `(Syntax.ident info $(quote rawVal) $(quote val) $(quote preresolved))
8755   -- Add global scopes at compilation time (now), add macro scope at runtime (in the quotation).
8756   -- See the paper for details.
8757   let r ← resolveGlobalName val
8758   -- extension of the paper algorithm: also store unique section variable names as top-level scopes
8759   -- so they can be captured and used inside the section, but not outside
8760   let r' := resolveSectionVariable (← read).sectionVars val
8761   let preresolved := r ++ r' ++ preresolved
8762   let val := quote val
8763   -- `scp` is bound in stxQuot.expand
8764   `(Syntax.ident info $(quote rawVal) (addMacroScope mainModule $val scp) $(quote preresolved))
8765 -- if antiquotation, insert contents as-is, else recurse
8766 | stx@(Syntax.node k _) => do
8767   if isAntiquot stx && !isEscapedAntiquot stx then
8768     getAntiquotTerm stx
8769   else if isTokenAntiquot stx && !isEscapedAntiquot stx then
8770     match stx[0] with
8771     | Syntax.atom _ val => `(Syntax.atom (Option.getD (getHeadInfo $(getAntiquotTerm stx)) info) $(quote val))
8772     | _                  => throwErrorAt stx "expected token"
8773   else if isAntiquotSuffixSplice stx && !isEscapedAntiquot stx then
8774     -- splices must occur in a `many` node
8775     throwErrorAt stx "unexpected antiquotation splice"
8776   else if isAntiquotSplice stx && !isEscapedAntiquot stx then
8777     throwErrorAt stx "unexpected antiquotation splice"
8778   else
8779     let empty ← `(Array.empty);
8780     -- if escaped antiquotation, decrement by one escape level
8781     let stx := unescapeAntiquot stx
8782     let args ← stx.getArgs.foldlM (fun args arg => do
8783       if k == nullKind && isAntiquotSuffixSplice arg then
8784         let antiquot := getAntiquotSuffixSpliceInner arg
8785         match antiquotSuffixSplice? arg with
8786         | `optional => `(Array.appendCore $args (match $(getAntiquotTerm antiquot):term with
8787           | some x => Array.empty.push x
8788           | none   => Array.empty))

```

```

8789 | `many      => `(Array.appendCore $args $(getAntiquotTerm antiquot))
8790 | `sepBy    => `(Array.appendCore $args (@SepArray.elimsAndSeps $(getSepFromSplice arg) $(getAntiquotTerm antiquot)))
8791 | k        => throwErrorAt arg "invalid antiquotation suffix splice kind '{k}'"
8792 else if k == nullKind && isAntiquotSplice arg then
8793   let k := antiquotSpliceKind? arg
8794   let (arg, bindLets) ← floatOutAntiquotTerms arg |>.run pure
8795   let inner ← (getAntiquotSpliceContents arg).mapM quoteSyntax
8796   let ids ← getAntiquotationIds arg
8797   if ids.isEmpty then
8798     throwErrorAt stx "antiquotation splice must contain at least one antiquotation"
8799   let arr ← match k with
8800   | `optional => `(match [$ids:ident],* with
8801   |   $[some $ids:ident],* => $(quote inner)
8802   |   none                => Array.empty)
8803   | _ =>
8804     let arr ← ids[:ids.size-1].foldrM (fun id arr => `(Array.zip $id $arr)) ids.back
8805     `(Array.map (fun $(← mkTuple ids) => $(inner[0])) $arr)
8806   let arr ←
8807     if k == `sepBy then
8808       `(mkSepArray $arr (mkAtom $(getSepFromSplice arg)))
8809     else arr
8810   let arr ← bindLets arr
8811   `(Array.appendCore $args $arr)
8812 else do
8813   let arg ← quoteSyntax arg;
8814   `(Array.push $args $arg)) empty
8815 `(Syntax.node $(quote k) $args)
8816 | Syntax.atom _ val =>
8817   `(Syntax.atom info $(quote val))
8818 | Syntax.missing => throwUnsupportedSyntax
8819
8820 def stxQuot.expand (stx : Syntax) : TermElabM Syntax := do
8821   /- Syntax quotations are monadic values depending on the current macro scope. For efficiency, we bind
8822   the macro scope once for each quotation, then build the syntax tree in a completely pure computation
8823   depending on this binding. Note that regular function calls do not introduce a new macro scope (i.e.
8824   we preserve referential transparency), so we can refer to this same `scp` inside `quoteSyntax` by
8825   including it literally in a syntax quotation. -/
8826   -- TODO: simplify to `(do scp ← getCurrMacroScope; pure $(quoteSyntax quoted))
8827   let stx ← quoteSyntax stx.getQuotContent;
8828   `(Bind.bind MonadRef.mkInfoFromRefPos (fun info =>
8829     Bind.bind getCurrMacroScope (fun scp =>
8830       Bind.bind getMainModule (fun mainModule => Pure.pure $stx))))
8831   /- NOTE: It may seem like the newly introduced binding `scp` may accidentally
8832   capture identifiers in an antiquotation introduced by `quoteSyntax`. However,
8833   note that the syntax quotation above enjoys the same hygiene guarantees as
8834   anywhere else in Lean; that is, we implement hygienic quotations by making
8835   use of the hygienic quotation support of the bootstrapped Lean compiler!

```

```

8836
8837 Aside: While this might sound "dangerous", it is in fact less reliant on a
8838 "chain of trust" than other bootstrapping parts of Lean: because this
8839 implementation itself never uses `scp` (or any other identifier) both inside
8840 and outside quotations, it can actually correctly be compiled by an
8841 unhygienic (but otherwise correct) implementation of syntax quotations. As
8842 long as it is then compiled again with the resulting executable (i.e. up to
8843 stage 2), the result is a correct hygienic implementation. In this sense the
8844 implementation is "self-stabilizing". It was in fact originally compiled
8845 by an unhygienic prototype implementation. -/
8846
8847 macro "elab_stx_quot" kind:ident : command =>
8848   `(@[builtinTermElab $kind:ident] def elabQuot : TermElab := adaptExpander stxQuot.expand)
8849
8850 --
8851
8852 elab_stx_quot Parser.Level.quot
8853 elab_stx_quot Parser.Term.quot
8854 elab_stx_quot Parser.Term.funBinder.quot
8855 elab_stx_quot Parser.Term.bracketedBinder.quot
8856 elab_stx_quot Parser.Term.matchDiscr.quot
8857 elab_stx_quot Parser.Tactic.quot
8858 elab_stx_quot Parser.Tactic.quotSeq
8859 elab_stx_quot Parser.Term.stx.quot
8860 elab_stx_quot Parser.Term.prec.quot
8861 elab_stx_quot Parser.Term.attr.quot
8862 elab_stx_quot Parser.Term.prio.quot
8863 elab_stx_quot Parser.Term.doElem.quot
8864 elab_stx_quot Parser.Term.dynamicQuot
8865
8866 /- match -/
8867
8868 -- an "alternative" of patterns plus right-hand side
8869 private abbrev Alt := List Syntax × Syntax
8870
8871 /--
8872 In a single match step, we match the first discriminant against the "head" of the first pattern of the first
8873 alternative. This datatype describes what kind of check this involves, which helps other patterns decide if
8874 they are covered by the same check and don't have to be checked again (see also `MatchResult`). -/
8875 inductive HeadCheck where
8876   -- match step that always succeeds: _, x, `($x), ...
8877   | unconditional
8878   -- match step based on kind and, optionally, arity of discriminant
8879   -- If `arity` is given, that number of new discriminants is introduced. `covered` patterns should then introduce the
8880   -- same number of new patterns.
8881   -- We actually check the arity at run time only in the case of `null` nodes since it should otherwise be implied by
8882   -- the node kind.

```

```

8883 -- without arity: `($x:k)
8884 -- with arity: any quotation without an antiquote head pattern
8885 | shape (k : SyntaxNodeKind) (arity : Option Nat)
8886 -- Match step that succeeds on `null` nodes of arity at least `numPrefix + numSuffix`, introducing discriminants
8887 -- for the first `numPrefix` children, one `null` node for those in between, and for the `numSuffix` last children.
8888 -- example: `([$x, $xs, *, $y]) is `slice 2 2`
8889 | slice (numPrefix numSuffix : Nat)
8890 -- other, complicated match step that will probably only cover identical patterns
8891 -- example: antiquote splices `($[...])*`
8892 | other (pat : Syntax)
8893
8894 open HeadCheck
8895
8896 /-- Describe whether a pattern is covered by a head check (induced by the pattern itself or a different pattern). -/
8897 inductive MatchResult where
8898   -- Pattern agrees with head check, remove and transform remaining alternative.
8899   -- If `exhaustive` is `false`, *also* include unchanged alternative in the "no" branch.
8900   | covered (f : Alt → TermElabM Alt) (exhaustive : Bool)
8901   -- Pattern disagrees with head check, include in "no" branch only
8902   | uncovered
8903   -- Pattern is not quite sure yet; include unchanged in both branches
8904   | undecided
8905
8906 open MatchResult
8907
8908 /-- All necessary information on a pattern head. -/
8909 structure HeadInfo where
8910   -- check induced by the pattern
8911   check : HeadCheck
8912   -- compute compatibility of pattern with given head check
8913   onMatch (taken : HeadCheck) : MatchResult
8914   -- actually run the specified head check, with the discriminant bound to `discr`
8915   doMatch (yes : (newDiscrs : List Syntax) → TermElabM Syntax) (no : TermElabM Syntax) : TermElabM Syntax
8916
8917 /-- Adapt alternatives that do not introduce new discriminants in `doMatch`, but are covered by those that do so. -/
8918 private def noOpMatchAdaptPats : HeadCheck → Alt → Alt
8919   | shape k (some sz), (pats, rhs) => (List.replicate sz (Unhygienic.run `(_)) ++ pats, rhs)
8920   | slice p s, (pats, rhs) => (List.replicate (p + 1 + s) (Unhygienic.run `(_)) ++ pats, rhs)
8921   | _, alt => alt
8922
8923 private def adaptRhs (fn : Syntax → TermElabM Syntax) : Alt → TermElabM Alt
8924   | (pats, rhs) => do (pats, ← fn rhs)
8925
8926 private partial def getHeadInfo (alt : Alt) : TermElabM HeadInfo :=
8927   let pat := alt.fst.head!
8928   let unconditionally (rhsFn) := pure {
8929     check := unconditional,

```

```

8930 doMatch := fun yes no => yes [],
8931 onMatch := fun taken => covered (adaptRhs rhsFn ◦ noOpMatchAdaptPats taken) (match taken with | unconditional => true | _ => false)
8932 }
8933 -- quotation pattern
8934 if isQuot pat then
8935   let quoted := getQuotContent pat
8936   if quoted.isAtom then
8937     -- We assume that atoms are uniquely determined by the node kind and never have to be checked
8938     unconditionally pure
8939   else if quoted.isTokenAntiquot then
8940     unconditionally `(let $(quoted.getAntiquotTerm) := discr; $(·))
8941   else if isAntiquot quoted && !isEscapedAntiquot quoted then
8942     -- quotation contains a single antiquotation
8943     let k := antiquotKind? quoted |>.get!
8944     match getAntiquotTerm quoted with
8945     | `(_) => unconditionally pure
8946     | `($id:ident) =>
8947       -- Antiquotation kinds like `$id:ident` influence the parser, but also need to be considered by
8948       -- `match` (but not by quotation terms). For example, `($id:ident)` and `($e)` are not
8949       -- distinguishable without checking the kind of the node to be captured. Note that some
8950       -- antiquotations like the latter one for terms do not correspond to any actual node kind
8951       -- (signified by `k == Name.anonymous`), so we would only check for `ident` here.
8952       --
8953       -- if stx.isOfKind `ident` then
8954       --   let id := stx; let e := stx; ...
8955       -- else
8956       --   let e := stx; ...
8957       let rhsFn := `(let $id := discr; $(·))
8958       if k == Name.anonymous then unconditionally rhsFn else pure {
8959         check := shape k none,
8960         onMatch := fun
8961           | taken@(shape k' sz) =>
8962             if k' == k then
8963               covered (adaptRhs rhsFn ◦ noOpMatchAdaptPats taken) (exhaustive := sz.isNone)
8964             else uncovered
8965           | _ => uncovered,
8966         doMatch := fun yes no => do `(cond (Syntax.isOfKind discr $(quote k)) $(← yes []) $(← no)),
8967       }
8968     | anti => throwErrorAt anti "unsupported antiquotation kind in pattern"
8969   else if isAntiquotSuffixSplice quoted then throwErrorAt quoted "unexpected antiquotation splice"
8970   else if isAntiquotSplice quoted then throwErrorAt quoted "unexpected antiquotation splice"
8971   else if quoted.getArgs.size == 1 && isAntiquotSuffixSplice quoted[0] then
8972     let anti := getAntiquotTerm (getAntiquotSuffixSpliceInner quoted[0])
8973     unconditionally fun rhs => match antiquotSuffixSplice? quoted[0] with
8974     | `optional => `(let $anti := Syntax.getOptional? discr; $rhs)
8975     | `many      => `(let $anti := Syntax.getArgs discr; $rhs)
8976     | `sepBy     => `(let $anti := @SepArray.mk $(getSepFromSplice quoted[0]) (Syntax.getArgs discr); $rhs)

```

```

8977 | k          => throwErrorAt quoted "invalid antiquotation suffix splice kind '{k}'"
8978 else if quoted.getArgs.size == 1 && isAntiquotSplice quoted[0] then pure {
8979   check    := other pat,
8980   onMatch  := fun
8981     | other pat' => if pat' == pat then covered pure (exhaustive := true) else undecided
8982     | _          => undecided,
8983   doMatch  := fun yes no => do
8984     let splice := quoted[0]
8985     let k := antiquotSpliceKind? splice
8986     let contents := getAntiquotSpliceContents splice
8987     let ids ← getAntiquotationIds splice
8988     let yes ← yes []
8989     let no ← no
8990     match k with
8991     | `optional =>
8992       let nones := mkArray ids.size (← `(none))
8993       `(let_delayed yes _ $ids* := $yes;
8994         if discr.isNone then yes () $[ $nones]*
8995         else match discr with
8996           | `($(mkNullNode contents)) => yes () $[ (some $ids)]*
8997           | _                        => $no)
8998     | _ =>
8999       let mut discrs ← `(Syntax.getArgs discr)
9000       if k == `sepBy then
9001         discrs ← `(Array.getSepElems $discrs)
9002       let tuple ← mkTuple ids
9003       let mut yes := yes
9004       let resId ← match ids with
9005         | #[id] => id
9006         | _      =>
9007           for id in ids do
9008             yes ← `(let $id := tuples.map (fun $tuple => $id); $yes)
9009           `(tuples)
9010       let contents := if contents.size == 1
9011         then contents[0]
9012         else mkNullNode contents
9013       `(match OptionM.run ($(discrs).sequenceMap fun
9014         | `($contents) => some $tuple
9015         | _             => none) with
9016         | some $resId => $yes
9017         | none        => $no)
9018 }
9019 else if let some idx := quoted.getArgs.findIdx? (fun arg => isAntiquotSuffixSplice arg || isAntiquotSplice arg) then do
9020   /-
9021     pattern of the form `match discr, ... with | `(pat_0 ... pat_(idx-1) $[...]* pat_(idx+1) ...), ...`
9022     transform to
9023     ```

```



```

9024     if discr.getNumArgs >= $quoted.getNumArgs - 1 then
9025         match discr[0], ..., discr[idx-1], mkNullNode (discr.getArgs.extract idx (discr.getNumArgs - $numSuffix)), ..., discr[quoted.!,
9026         ... | `(pat_0), ... `(pat_(idx-1)), `($[...])*`, `(pat_(idx+1)), ...
9027
9028     -/
9029     let numSuffix := quoted.getNumArgs - 1 - idx
9030     pure {
9031         check := slice idx numSuffix
9032         onMatch := fun
9033             | slice p s =>
9034                 if p == idx && s == numSuffix then
9035                     let argPats := quoted.getArgs.mapIdx fun i arg =>
9036                         let arg := if (i : Nat) == idx then mkNullNode #[arg] else arg
9037                         Unhygienic.run `(`($arg))
9038                     covered (fun (pats, rhs) => (argPats.toList ++ pats, rhs)) (exhaustive := true)
9039                 else uncovered
9040             | _ => uncovered
9041     doMatch := fun yes no => do
9042         let prefixDiscrs ← (List.range idx).mapM `(`(Syntax.getArg discr $(quote .)))
9043         let sliceDiscr ← `(mkNullNode (discr.getArgs.extract $(quote idx) (discr.getNumArgs - $(quote numSuffix))))
9044         let suffixDiscrs ← (List.range numSuffix).mapM fun i =>
9045             `(Syntax.getArg discr (discr.getNumArgs - $(quote (numSuffix - i))))
9046         `(ite (GE.ge discr.getNumArgs $(quote (quoted.getNumArgs - 1)))
9047             $(← yes (prefixDiscrs ++ sliceDiscr :: suffixDiscrs))
9048             $(← no))
9049     }
9050 else
9051     -- not an antiquotation, or an escaped antiquotation: match head shape
9052     let quoted := unescapeAntiquot quoted
9053     let kind := quoted.getKind
9054     let argPats := quoted.getArgs.map fun arg => Unhygienic.run `(`($arg))
9055     pure {
9056         check := shape kind argPats.size,
9057         onMatch := fun taken =>
9058             if (match taken with | shape k' sz => k' == kind && sz == argPats.size | _ => false : Bool) then
9059                 covered (fun (pats, rhs) => (argPats.toList ++ pats, rhs)) (exhaustive := true)
9060             else
9061                 uncovered,
9062         doMatch := fun yes no => do
9063             let cond ← match kind with
9064             | `null => `(Syntax.matchesNull discr $(quote argPats.size))
9065             | `ident => `(Syntax.matchesIdent discr $(quote quoted.getId))
9066             | _ => `(Syntax.isOfKind discr $(quote kind))
9067             let newDiscrs ← (List.range argPats.size).mapM fun i => `(Syntax.getArg discr $(quote i))
9068             `(ite (Eq $cond true) $(← yes newDiscrs) $(← no))
9069     }
9070 else match pat with

```

```

9071 | `(_)                => unconditionally pure
9072 | `($id:ident)        => unconditionally `(let $id := discr; $(.)))
9073 | `($id:ident@$pat) => do
9074   let info ← getHeadInfo (pat::alt.1.tail!, alt.2)
9075   { info with onMatch := fun taken => match info.onMatch taken with
9076     | covered f exh => covered (fun alt => f alt >>= adaptRhs `(let $id := discr; $(.))) exh
9077     | r              => r }
9078 | _                  => throwErrorAt pat "match_syntax: unexpected pattern kind {pat}"
9079
9080 -- Bind right-hand side to new `let_delayed` decl in order to prevent code duplication
9081 private def deduplicate (floatedLetDecls : Array Syntax) : Alt → TermElabM (Array Syntax × Alt)
9082 -- NOTE: new macro scope so that introduced bindings do not collide
9083 | (pats, rhs) => do
9084   if let `($f:ident $[ $args:ident]*) := rhs then
9085     -- looks simple enough/created by this function, skip
9086     return (floatedLetDecls, (pats, rhs))
9087   withFreshMacroScope do
9088     match ← getPatternsVars pats.toArray with
9089     | #[] =>
9090       -- no antiquotations => introduce Unit parameter to preserve evaluation order
9091       let rhs' ← `(rhs Unit.unit)
9092       (floatedLetDecls.push (← `(letDecl|rhs _ := $rhs)), (pats, rhs'))
9093     | vars =>
9094       let rhs' ← `(rhs $vars*)
9095       (floatedLetDecls.push (← `(letDecl|rhs $vars:ident* := $rhs)), (pats, rhs'))
9096
9097 private partial def compileStxMatch (discrs : List Syntax) (alts : List Alt) : TermElabM Syntax := do
9098   trace[Elab.match_syntax] "match {discrs} with {alts}"
9099   match discrs, alts with
9100   | [],          ([], rhs)::_ => pure rhs -- nothing left to match
9101   | _,          []           => throwError "non-exhaustive 'match_syntax'"
9102   | discr::discrs, alt::alts => do
9103     let info ← getHeadInfo alt
9104     let pat := alt.1.head!
9105     let alts ← (alt::alts).mapM fun alt => do ((← getHeadInfo alt).onMatch info.check, alt)
9106     let mut yesAlts := #[]
9107     let mut undecidedAlts := #[]
9108     let mut nonExhaustiveAlts := #[]
9109     let mut floatedLetDecls := #[]
9110     for alt in alts do
9111       let mut alt := alt
9112       match alt with
9113       | (covered f exh, alt) =>
9114         -- we can only factor out a common check if there are no undecided patterns in between;
9115         -- otherwise we would change the order of alternatives
9116         if undecidedAlts.isEmpty then
9117           yesAlts ← yesAlts.push <$> f (alt.1.tail!, alt.2)

```

```

9118         if !exh then
9119             nonExhaustiveAlts := nonExhaustiveAlts.push alt
9120         else
9121             (floatedLetDecls, alt) ← deduplicate floatedLetDecls alt
9122             undecidedAlts := undecidedAlts.push alt
9123             nonExhaustiveAlts := nonExhaustiveAlts.push alt
9124     | (undecided, alt) =>
9125         (floatedLetDecls, alt) ← deduplicate floatedLetDecls alt
9126         undecidedAlts := undecidedAlts.push alt
9127         nonExhaustiveAlts := nonExhaustiveAlts.push alt
9128     | (uncovered, alt) =>
9129         nonExhaustiveAlts := nonExhaustiveAlts.push alt
9130 let mut stx ← info.doMatch
9131 (yes := fun newDiscrs => do
9132     let mut yesAlts := yesAlts
9133     if !undecidedAlts.isEmpty then
9134         -- group undecided alternatives in a new default case `| discr2, ... => match discr, discr2, ... with ...`
9135         let vars ← discrs.mapM fun _ => withFreshMacroScope `(discr)
9136         let pats := List.replicate newDiscrs.length (Unhygienic.run `(_)) ++ vars
9137         let alts ← undecidedAlts.mapM fun alt => `(matchAltExpr | | $(alt.1.toArray),* => $(alt.2))
9138         let rhs ← `(match discr, $(vars.toArray):term],* with $alts:matchAlt*)
9139         yesAlts := yesAlts.push (pats, rhs)
9140         withFreshMacroScope $ compileStxMatch (newDiscrs ++ discrs) yesAlts.toList)
9141     (no := withFreshMacroScope $ compileStxMatch (discr::discrs) nonExhaustiveAlts.toList)
9142 for d in floatedLetDecls do
9143     stx ← `(let_delayed $d:letDecl; $stx)
9144 `(let discr := $discr; $stx)
9145 | _, _ => unreachable!
9146
9147 def match_syntax.expand (stx : Syntax) : TermElabM Syntax := do
9148     match stx with
9149     | `(match $[discrs:term],* with $[ patss ],* => $rhss]*) => do
9150         if !patss.any (·.any (fun
9151             | `($id$pat) => pat.isQuot
9152             | pat      => pat.isQuot)) then
9153             -- no quotations => fall back to regular `match`
9154             throwUnsupportedSyntax
9155         let stx ← compileStxMatch discrs.toList (patss.map (·.toList) |>.zip rhss).toList
9156         trace[Elab.match_syntax.result] "{stx}"
9157         stx
9158     | _ => throwUnsupportedSyntax
9159
9160 @[builtinTermElab «match»] def elabMatchSyntax : TermElab :=
9161     adaptExpander match_syntax.expand
9162
9163 builtin_initialize
9164     registerTraceClass `Elab.match_syntax

```

```

9165   registerTraceClass `Elab.match_syntax.result
9166
9167 end Lean.Elab.Term.Quotation
9168 ::::::::::::::
9169 Elab/SetOption.lean
9170 ::::::::::::::
9171 /-
9172 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
9173 Released under Apache 2.0 license as described in the file LICENSE.
9174 Authors: Leonardo de Moura
9175 -/
9176 import Lean.Elab.Log
9177 import Lean.Elab.InfoTree
9178 namespace Lean.Elab
9179
9180 variable [Monad m] [MonadOptions m] [MonadExceptOf Exception m] [MonadRef m]
9181 variable [AddErrorMessageContext m] [MonadLiftT (EIO Exception) m] [MonadInfoTree m]
9182
9183 def elabSetOption (id : Syntax) (val : Syntax) : m Options := do
9184   let optionName := id.getId.eraseMacroScopes
9185   match val.isStrLit? with
9186   | some str => setOption optionName (DataValue.ofString str)
9187   | none     =>
9188     match val.isNatLit? with
9189     | some num => setOption optionName (DataValue.ofNat num)
9190     | none     =>
9191       match val with
9192       | Syntax.atom _ "true"  => setOption optionName (DataValue.ofBool true)
9193       | Syntax.atom _ "false" => setOption optionName (DataValue.ofBool false)
9194       | _ =>
9195         addCompletionInfo <| CompletionInfo.option (← getRef)
9196         throwError "unexpected set_option value {val}"
9197 where
9198   setOption (optionName : Name) (val : DataValue) : m Options := do
9199     let ref ← getRef
9200     let decl ← IO.toEIO (fun (ex : IO.Error) => Exception.error ref ex.toString) (getOptionDecl optionName)
9201     unless decl.defValue.sameCtor val do throwError "type mismatch at set_option"
9202     return (← getOptions).insert optionName val
9203
9204 end Lean.Elab
9205 ::::::::::::::
9206 Elab/StructInst.lean
9207 ::::::::::::::
9208 /-
9209 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
9210 Released under Apache 2.0 license as described in the file LICENSE.
9211 Authors: Leonardo de Moura

```

```

9212 -/
9213 import Lean.Util.FindExpr
9214 import Lean.Parser.Term
9215 import Lean.Elab.App
9216 import Lean.Elab.Binders
9217
9218 namespace Lean.Elab.Term.StructInst
9219
9220 open Std (HashMap)
9221 open Meta
9222
9223 /-
9224   Structure instances are of the form:
9225
9226       "{" >> optional (atomic (termParser >> " with "))
9227       >> many1Indent (group (structInstField >> optional ", "))
9228       >> optEllipsis
9229       >> optional (" : " >> termParser)
9230       >> "}"
9231 -/
9232
9233 @[builtinMacro Lean.Parser.Term.structInst] def expandStructInstExpectedType : Macro := fun stx =>
9234   let expectedArg := stx[4]
9235   if expectedArg.isNone then
9236     Macro.throwUnsupported
9237   else
9238     let expected := expectedArg[1]
9239     let stxNew := stx.setArg 4 mkNullNode
9240     `(($stxNew : $expected))
9241
9242 /-
9243   If `stx` is of the form `{ s with ... }` and `s` is not a local variable, expand into `let src := s; { src with ... }`.
9244
9245   Note that this one is not a `Macro` because we need to access the local context.
9246 -/
9247 private def expandNonAtomicExplicitSource (stx : Syntax) : TermElabM (Option Syntax) :=
9248   withFreshMacroScope do
9249     let sourceOpt := stx[1]
9250     if sourceOpt.isNone then
9251       pure none
9252     else
9253       let source := sourceOpt[0]
9254       match (← isLocalIdent? source) with
9255       | some _ => pure none
9256       | none   =>
9257         if source.isMissing then
9258           throwAbortTerm

```

```

9259     else
9260         let src ← `(src)
9261         let sourceOpt := sourceOpt.setArg 0 src
9262         let stxNew    := stx.setArg 1 sourceOpt
9263         `(let src := $source; $stxNew)
9264
9265 inductive Source where
9266   | none      -- structure instance source has not been provided
9267   | implicit (stx : Syntax) -- `..`
9268   | explicit (stx : Syntax) (src : Expr) -- `src with`
9269   deriving Inhabited
9270
9271 def Source.isNone : Source → Bool
9272   | Source.none => true
9273   | _           => false
9274
9275 def setStructSourceSyntax (structStx : Syntax) : Source → Syntax
9276   | Source.none      => (structStx.setArg 1 mkNullNode).setArg 3 mkNullNode
9277   | Source.implicit stx => (structStx.setArg 1 mkNullNode).setArg 3 stx
9278   | Source.explicit stx _ => (structStx.setArg 1 stx).setArg 3 mkNullNode
9279
9280 private def getStructSource (stx : Syntax) : TermElabM Source :=
9281   withRef stx do
9282     let explicitSource := stx[1]
9283     let implicitSource := stx[3]
9284     if explicitSource.isNone && implicitSource[0].isNone then
9285       return Source.none
9286     else if explicitSource.isNone then
9287       return Source.implicit implicitSource
9288     else if implicitSource[0].isNone then
9289       let fvar? ← isLocalIdent? explicitSource[0]
9290       match fvar? with
9291       | none      => unreachable! -- expandNonAtomicExplicitSource must have been used when we get here
9292       | some src  => return Source.explicit explicitSource src
9293     else
9294       throwError "invalid structure instance `with` and `..` cannot be used together"
9295
9296 /-
9297   We say a `{ ... }` notation is a `modifyOp` if it contains only one
9298   ```
9299   def structInstArrayRef := leading_parser "[" >> termParser >> "]"
9300   ```
9301 -/
9302 private def isModifyOp? (stx : Syntax) : TermElabM (Option Syntax) := do
9303   let s? ← stx[2].getArgs.foldlM (init := none) fun s? p =>
9304     /- p is of the form `(group (structInstField >> optional ", "))` -/
9305     let arg := p[0]

```

```

9306  /- Remark: the syntax for `structInstField` is
9307  `\"
9308  def structInstLVal  := leading_parser (ident <|> numLit <|> structInstArrayRef) >> many (group (\" >> (ident <|> numLit)) <|> s
9309  def structInstField := leading_parser structInstLVal >> \" := \" >> termParser
9310  `\"
9311  -/
9312  let lval := arg[0]
9313  let k    := lval[0].getKind
9314  if k == `Lean.Parser.Term.structInstArrayRef then
9315    match s? with
9316    | none    => pure (some arg)
9317    | some s =>
9318      if s.getKind == `Lean.Parser.Term.structInstArrayRef then
9319        throwErrorAt arg \"invalid \{...\} notation, at most one `[..]` at a given level\"
9320      else
9321        throwErrorAt arg \"invalid \{...\} notation, can't mix field and `[..]` at a given level\"
9322  else
9323    match s? with
9324    | none    => pure (some arg)
9325    | some s =>
9326      if s.getKind == `Lean.Parser.Term.structInstArrayRef then
9327        throwErrorAt arg \"invalid \{...\} notation, can't mix field and `[..]` at a given level\"
9328      else
9329        pure s?
9330  match s? with
9331  | none    => pure none
9332  | some s => if s[0][0].getKind == `Lean.Parser.Term.structInstArrayRef then pure s? else pure none
9333
9334 private def elabModifyOp (stx modifyOp source : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
9335   let cont (val : Syntax) : TermElabM Expr := do
9336     let lval := modifyOp[0][0]
9337     let idx  := lval[1]
9338     let self := source[0]
9339     let stxNew ← `($(self).modifyOp (idx := $idx) (fun s => $val))
9340     trace[Elab.struct.modifyOp] \"{stx}\\n====>\\n{stxNew}\"
9341     withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
9342   trace[Elab.struct.modifyOp] \"{modifyOp}\\nSource: {source}\"
9343   let rest := modifyOp[0][1]
9344   if rest.isNone then
9345     cont modifyOp[2]
9346   else
9347     let s ← `(s)
9348     let valFirst := rest[0]
9349     let valFirst := if valFirst.getKind == `Lean.Parser.Term.structInstArrayRef then valFirst else valFirst[1]
9350     let restArgs := rest.getArgs
9351     let valRest  := mkNullNode restArgs[1:restArgs.size]
9352     let valField := modifyOp.setArg 0 <| Syntax.node ``Parser.Term.structInstLVal #[valFirst, valRest]

```

```

9353   let valSource := source.modifyArg 0 fun _ => s
9354   let val       := stx.setArg 1 valSource
9355   let val       := val.setArg 2 <| mkNullNode #[mkNullNode #[valField, mkNullNode]]
9356   trace[Elab.struct.modifyOp] "{stx}\nval: {val}"
9357   cont val
9358
9359   /- Get structure name and elaborate explicit source (if available) -/
9360 private def getStructName (stx : Syntax) (expectedType? : Option Expr) (sourceView : Source) : TermElabM (Name × Expr) := do
9361   tryPostponeIfNoneOrMVar expectedType?
9362   let useSource : Unit → TermElabM (Name × Expr) := fun _ =>
9363     match sourceView, expectedType? with
9364     | Source.explicit _ src, _ => do
9365       let srcType ← inferType src
9366       let srcType ← whnf srcType
9367       tryPostponeIfMVar srcType
9368       match srcType.getAppFn with
9369       | Expr.const constName _ _ => return (constName, srcType)
9370       | _ => throwUnexpectedExpectedType srcType "source"
9371     | _, some expectedType => throwUnexpectedExpectedType expectedType
9372     | _, none             => throwUnknownExpectedType
9373   match expectedType? with
9374   | none => useSource ()
9375   | some expectedType =>
9376     let expectedType ← whnf expectedType
9377     match expectedType.getAppFn with
9378     | Expr.const constName _ _ => return (constName, expectedType)
9379     | _                        => useSource ()
9380 where
9381   throwUnknownExpectedType :=
9382     throwError "invalid \{...} notation, expected type is not known"
9383   throwUnexpectedExpectedType type (kind := "expected") := do
9384     let type ← instantiateMVars type
9385     if type.getAppFn.isMVar then
9386       throwUnknownExpectedType
9387     else
9388       throwError "invalid \{...} notation, {kind} type is not of the form (C ...){indentExpr type}"
9389
9390 inductive FieldLHS where
9391   | fieldName (ref : Syntax) (name : Name)
9392   | fieldIndex (ref : Syntax) (idx : Nat)
9393   | modifyOp (ref : Syntax) (index : Syntax)
9394   deriving Inhabited
9395
9396 instance : ToFormat FieldLHS := (fun lhs =>
9397   match lhs with
9398   | FieldLHS.fieldName _ n => fmt n
9399   | FieldLHS.fieldIndex _ i => fmt i

```



```

9400 | FieldLHS.modifyOp _ i => "[" ++ i.prettyPrint ++ "]"
9401
9402 inductive FieldVal (σ : Type) where
9403 | term (stx : Syntax) : FieldVal σ
9404 | nested (s : σ)      : FieldVal σ
9405 | default             : FieldVal σ -- mark that field must be synthesized using default value
9406 deriving Inhabited
9407
9408 structure Field (σ : Type) where
9409   ref : Syntax
9410   lhs : List FieldLHS
9411   val : FieldVal σ
9412   expr? : Option Expr := none
9413   deriving Inhabited
9414
9415 def Field.isSimple {σ} : Field σ → Bool
9416 | { lhs := [_], .. } => true
9417 | _                  => false
9418
9419 inductive Struct where
9420 | mk (ref : Syntax) (structName : Name) (fields : List (Field Struct)) (source : Source)
9421   deriving Inhabited
9422
9423 abbrev Fields := List (Field Struct)
9424
9425 /- true if all fields of the given structure are marked as `default` -/
9426 partial def Struct.allDefault : Struct → Bool
9427 | (ref, _, fields, _) => fields.all fun (ref, _, val, _) => match val with
9428 | FieldVal.term _    => false
9429 | FieldVal.default   => true
9430 | FieldVal.nested s  => allDefault s
9431
9432 def Struct.ref : Struct → Syntax
9433 | (ref, _, _, _) => ref
9434
9435 def Struct.structName : Struct → Name
9436 | (ref, structName, _, _) => structName
9437
9438 def Struct.fields : Struct → Fields
9439 | (ref, _, fields, _) => fields
9440
9441 def Struct.source : Struct → Source
9442 | (ref, _, _, s) => s
9443
9444 def formatField (formatStruct : Struct → Format) (field : Field Struct) : Format :=
9445   Format.joinSep field.lhs " . " ++ " := " ++
9446   match field.val with

```

```

9447 | FieldVal.term v    => v.prettyPrint
9448 | FieldVal.nested s => formatStruct s
9449 | FieldVal.default  => "<default>"
9450
9451 partial def formatStruct : Struct → Format
9452 | (_, structName, fields, source) =>
9453   let fieldsFmt := Format.joinSep (fields.map (formatField formatStruct)) ", "
9454   match source with
9455   | Source.none           => "{" ++ fieldsFmt ++ "}"
9456   | Source.implicit _    => "{" ++ fieldsFmt ++ " .. }"
9457   | Source.explicit _ src => "{" ++ format src ++ " with " ++ fieldsFmt ++ "}"
9458
9459 instance : ToFormat Struct      := {formatStruct}
9460 instance : ToString Struct := {toString ∘ format}
9461
9462 instance : ToFormat (Field Struct) := {formatField formatStruct}
9463 instance : ToString (Field Struct) := {toString ∘ format}
9464
9465 /-
9466 Recall that `structInstField` elements have the form
9467 ```
9468   def structInstField := leading_parser structInstLVal >> " := " >> termParser
9469   def structInstLVal  := leading_parser (ident <|> numLit <|> structInstArrayRef) >> many (("." >> (ident <|> numLit)) <|> structInstA
9470   def structInstArrayRef := leading_parser "[" >> termParser >> "]"
9471 ```
9472 -/
9473 -- Remark: this code relies on the fact that `expandStruct` only transforms `fieldLHS.fieldName`
9474 def FieldLHS.toSyntax (first : Bool) : FieldLHS → Syntax
9475 | FieldLHS.modifyOp stx _    => stx
9476 | FieldLHS.fieldName stx name => if first then mkIdentFrom stx name else mkGroupNode #[mkAtomFrom stx ".", mkIdentFrom stx name]
9477 | FieldLHS.fieldIndex stx _  => if first then stx else mkGroupNode #[mkAtomFrom stx ".", stx]
9478
9479 def FieldVal.toSyntax : FieldVal Struct → Syntax
9480 | FieldVal.term stx => stx
9481 | _                 => unreachable!
9482
9483 def Field.toSyntax : Field Struct → Syntax
9484 | field =>
9485   let stx := field.ref
9486   let stx := stx.setArg 2 field.val.toSyntax
9487   match field.lhs with
9488   | first::rest => stx.setArg 0 <| mkNullNode #[first.toSyntax true, mkNullNode <| rest.toArray.map (FieldLHS.toSyntax false) ]
9489   | _          => unreachable!
9490
9491 private def toFieldLHS (stx : Syntax) : Except String FieldLHS :=
9492   if stx.getKind == `Lean.Parser.Term.structInstArrayRef then
9493     return FieldLHS.modifyOp stx stx[1]

```

```

9494 else
9495   -- Note that the representation of the first field is different.
9496   let stx := if stx.getKind == groupKind then stx[1] else stx
9497   if stx.isIdent then
9498     return FieldLHS.fieldName stx stx.getId.eraseMacroScopes
9499   else match stx.isFieldIdx? with
9500   | some idx => return FieldLHS.fieldIndex stx idx
9501   | none     => throw "unexpected structure syntax"
9502
9503 private def mkStructView (stx : Syntax) (structName : Name) (source : Source) : Except String Struct := do
9504   /- Recall that `stx` is of the form
9505   ```
9506   leading_parser "{" >> optional (atomic (termParser >> " with "))
9507                   >> many1Indent (group (structInstField >> optional ", "))
9508                   >> optional ".."
9509                   >> optional (" : " >> termParser)
9510                   >> "}"
9511   ```
9512   -/
9513   let fieldsStx := stx[2].getArgs.map (·[0])
9514   let fields ← fieldsStx.toList.mapM fun fieldStx => do
9515     let val    := fieldStx[2]
9516     let first ← toFieldLHS fieldStx[0][0]
9517     let rest  ← fieldStx[0][1].getArgs.toList.mapM toFieldLHS
9518     pure { ref := fieldStx, lhs := first :: rest, val := FieldVal.term val : Field Struct }
9519   pure (stx, structName, fields, source)
9520
9521 def Struct.modifyFieldsM {m : Type → Type} [Monad m] (s : Struct) (f : Fields → m Fields) : m Struct :=
9522   match s with
9523   | (ref, structName, fields, source) => return (ref, structName, (← f fields), source)
9524
9525 @[inline] def Struct.modifyFields (s : Struct) (f : Fields → Fields) : Struct :=
9526   Id.run <| s.modifyFieldsM f
9527
9528 def Struct.setFields (s : Struct) (fields : Fields) : Struct :=
9529   s.modifyFields fun _ => fields
9530
9531 private def expandCompositeFields (s : Struct) : Struct :=
9532   s.modifyFields fun fields => fields.map fun field => match field with
9533   | { lhs := FieldLHS.fieldName ref (Name.str Name.anonymous _) :: rest, .. } => field
9534   | { lhs := FieldLHS.fieldName ref n@(Name.str _ _) :: rest, .. } =>
9535     let newEntries := n.components.map <| FieldLHS.fieldName ref
9536     { field with lhs := newEntries ++ rest }
9537   | _ => field
9538
9539 private def expandNumLitFields (s : Struct) : TermElabM Struct :=
9540   s.modifyFieldsM fun fields => do

```

```

9541   let env ← getEnv
9542   let fieldNames := getStructureFields env s.structName
9543   fields.mapM fun field => match field with
9544   | { lhs := FieldLHS.fieldIndex ref idx :: rest, .. } =>
9545     if idx == 0 then throwErrorAt ref "invalid field index, index must be greater than 0"
9546     else if idx > fieldNames.size then throwErrorAt ref "invalid field index, structure has only #{fieldNames.size} fields"
9547     else pure { field with lhs := FieldLHS.fieldName ref fieldNames[idx - 1] :: rest }
9548   | _ => pure field
9549
9550 /- For example, consider the following structures:
9551 ```
9552   structure A where
9553     x : Nat
9554
9555   structure B extends A where
9556     y : Nat
9557
9558   structure C extends B where
9559     z : Bool
9560 ```
9561 This method expands parent structure fields using the path to the parent structure.
9562 For example,
9563 ```
9564 { x := 0, y := 0, z := true : C }
9565 ```
9566 is expanded into
9567 ```
9568 { toB.toA.x := 0, toB.y := 0, z := true : C }
9569 ```
9570 -/
9571 private def expandParentFields (s : Struct) : TermElabM Struct := do
9572   let env ← getEnv
9573   s.modifyFieldsM fun fields => fields.mapM fun field => match field with
9574   | { lhs := FieldLHS.fieldName ref fieldName :: rest, .. } =>
9575     match findField? env s.structName fieldName with
9576     | none => throwErrorAt ref "'{fieldName}' is not a field of structure '{s.structName}'"
9577     | some baseStructName =>
9578       if baseStructName == s.structName then pure field
9579       else match getPathToBaseStructure? env baseStructName s.structName with
9580       | some path => do
9581         let path := path.map fun funName => match funName with
9582         | Name.str _ s _ => FieldLHS.fieldName ref (Name.mkSimple s)
9583         | _ => unreachable!
9584         pure { field with lhs := path ++ field.lhs }
9585       | _ => throwErrorAt ref "failed to access field '{fieldName}' in parent structure"
9586   | _ => pure field
9587

```

```

9588 private abbrev FieldMap := HashMap Name Fields
9589
9590 private def mkFieldMap (fields : Fields) : TermElabM FieldMap :=
9591   fields.foldlM (init := {}) fun fieldMap field =>
9592     match field.lhs with
9593     | FieldLHS.fieldName _ fieldName :: rest =>
9594       match fieldMap.find? fieldName with
9595       | some (prevField::restFields) =>
9596         if field.isSimple || prevField.isSimple then
9597           throwErrorAt field.ref "field '{fieldName}' has already been specified"
9598         else
9599           return fieldMap.insert fieldName (field::prevField::restFields)
9600       | _ => return fieldMap.insert fieldName [field]
9601     | _ => unreachable!
9602
9603 private def isSimpleField? : Fields → Option (Field Struct)
9604   | [field] => if field.isSimple then some field else none
9605   | _       => none
9606
9607 private def getFieldIdx (structName : Name) (fieldNames : Array Name) (fieldName : Name) : TermElabM Nat := do
9608   match fieldNames.findIdx? fun n => n == fieldName with
9609   | some idx => pure idx
9610   | none     => throwError "field '{fieldName}' is not a valid field of '{structName}'"
9611
9612 private def mkProjStx (s : Syntax) (fieldName : Name) : Syntax :=
9613   Syntax.node `Lean.Parser.Term.proj #[s, mkAtomFrom s ".", mkIdentFrom s fieldName]
9614
9615 private def mkSubstructSource (structName : Name) (fieldNames : Array Name) (fieldName : Name) (src : Source) : TermElabM Source :=
9616   match src with
9617   | Source.explicit stx src => do
9618     let idx ← getFieldIdx structName fieldNames fieldName
9619     let stx := stx.modifyArg 0 fun stx => mkProjStx stx fieldName
9620     return Source.explicit stx (mkProj structName idx src)
9621   | s => return s
9622
9623 @[specialize] private def groupFields (expandStruct : Struct → TermElabM Struct) (s : Struct) : TermElabM Struct := do
9624   let env ← getEnv
9625   let fieldNames := getStructureFields env s.structName
9626   withRef s.ref do
9627     s.modifyFieldsM fun fields => do
9628       let fieldMap ← mkFieldMap fields
9629       fieldMap.toList.mapM fun (fieldName, fields) => do
9630         match isSimpleField? fields with
9631         | some field => pure field
9632         | none       =>
9633           let substructFields := fields.map fun field => { field with lhs := field.lhs.tail! }
9634           let substructSource ← mkSubstructSource s.structName fieldNames fieldName s.source

```

```

9635     let field := fields.head!
9636     match Lean.isSubobjectField? env s.structName fieldName with
9637     | some substructName =>
9638         let substruct := Struct.mk s.ref substructName substructFields substructSource
9639         let substruct ← expandStruct substruct
9640         pure { field with lhs := [field.lhs.head!], val := FieldVal.nested substruct }
9641     | none => do
9642         -- It is not a substructure field. Thus, we wrap fields using `Syntax`, and use `elabTerm` to process them.
9643         let valStx := s.ref -- construct substructure syntax using s.ref as template
9644         let valStx := valStx.setArg 4 mkNullNode -- erase optional expected type
9645         let args := substructFields.toArray.map fun field => mkNullNode #[field.toSyntax, mkNullNode]
9646         let valStx := valStx.setArg 2 (mkNullNode args)
9647         let valStx := setStructSourceSyntax valStx substructSource
9648         pure { field with lhs := [field.lhs.head!], val := FieldVal.term valStx }
9649
9650 def findField? (fields : Fields) (fieldName : Name) : Option (Field Struct) :=
9651   fields.find? fun field =>
9652     match field.lhs with
9653     | [FieldLHS.fieldName _ n] => n == fieldName
9654     | _ => false
9655
9656 @[specialize] private def addMissingFields (expandStruct : Struct → TermElabM Struct) (s : Struct) : TermElabM Struct := do
9657   let env ← getEnv
9658   let fieldNames := getStructureFields env s.structName
9659   let ref := s.ref
9660   withRef ref do
9661     let fields ← fieldNames.foldlM (init := []) fun fields fieldName => do
9662       match findField? s.fields fieldName with
9663       | some field => return field::fields
9664       | none =>
9665         let addField (val : FieldVal Struct) : TermElabM Fields := do
9666           return { ref := s.ref, lhs := [FieldLHS.fieldName s.ref fieldName], val := val } :: fields
9667         match Lean.isSubobjectField? env s.structName fieldName with
9668         | some substructName => do
9669           let substructSource ← mkSubstructSource s.structName fieldNames fieldName s.source
9670           let substruct := Struct.mk s.ref substructName [] substructSource
9671           let substruct ← expandStruct substruct
9672           addField (FieldVal.nested substruct)
9673         | none =>
9674           match s.source with
9675           | Source.none => addField FieldVal.default
9676           | Source.implicit _ => addField (FieldVal.term (mkHole s.ref))
9677           | Source.explicit stx _ =>
9678             -- stx is of the form `optional (try (termParser >> "with"))`
9679             let src := stx[0]
9680             let val := mkProjStx src fieldName
9681             addField (FieldVal.term val)

```

```

9682     return s.setFields fields.reverse
9683
9684 private partial def expandStruct (s : Struct) : TermElabM Struct := do
9685   let s := expandCompositeFields s
9686   let s ← expandNumLitFields s
9687   let s ← expandParentFields s
9688   let s ← groupFields expandStruct s
9689   addMissingFields expandStruct s
9690
9691 structure CtorHeaderResult where
9692   ctorFn      : Expr
9693   ctorFnType  : Expr
9694   instMVars   : Array MVarId := #[]
9695
9696 private def mkCtorHeaderAux : Nat → Expr → Expr → Array MVarId → TermElabM CtorHeaderResult
9697 | 0, type, ctorFn, instMVars => pure { ctorFn := ctorFn, ctorFnType := type, instMVars := instMVars }
9698 | n+1, type, ctorFn, instMVars => do
9699   let type ← whnfForall type
9700   match type with
9701 | Expr.forallE _ d b c =>
9702   match c.binderInfo with
9703 | BinderInfo.instImplicit =>
9704   let a ← mkFreshExprMVar d MetavarKind.synthetic
9705   mkCtorHeaderAux n (b.instantiate1 a) (mkApp ctorFn a) (instMVars.push a.mvarId!)
9706 | _ =>
9707   let a ← mkFreshExprMVar d
9708   mkCtorHeaderAux n (b.instantiate1 a) (mkApp ctorFn a) instMVars
9709 | _ => throwError "unexpected constructor type"
9710
9711 private partial def getForallBody : Nat → Expr → Option Expr
9712 | i+1, Expr.forallE _ _ b _ => getForallBody i b
9713 | i+1, _ => none
9714 | 0, type => type
9715
9716 private def propagateExpectedType (type : Expr) (numFields : Nat) (expectedType? : Option Expr) : TermElabM Unit :=
9717 match expectedType? with
9718 | none => pure ()
9719 | some expectedType => do
9720   match getForallBody numFields type with
9721   | none => pure ()
9722   | some typeBody =>
9723     unless typeBody.hasLooseBVars do
9724       discard <| isDefEq expectedType typeBody
9725
9726 private def mkCtorHeader (ctorVal : ConstructorVal) (expectedType? : Option Expr) : TermElabM CtorHeaderResult := do
9727   let us ← mkFreshLevelMVars ctorVal.levelParams.length
9728   let val := Lean.mkConst ctorVal.name us

```

```

9729 let type := (ConstantInfo.ctorInfo ctorVal).instantiateTypeLevelParams us
9730 let r ← mkCtorHeaderAux ctorVal.numParams type val #[]
9731 propagateExpectedType r.ctorFnType ctorVal.numFields expectedType?
9732 synthesizeAppInstMVars r.instMVars
9733 pure r
9734
9735 def markDefaultMissing (e : Expr) : Expr :=
9736   mkAnnotation `structInstDefault e
9737
9738 def defaultMissing? (e : Expr) : Option Expr :=
9739   annotation? `structInstDefault e
9740
9741 def throwFailedToElabField {α} (fieldName : Name) (structName : Name) (msgData : MessageData) : TermElabM α :=
9742   throwError "failed to elaborate field '{fieldName}' of '{structName}', {msgData}"
9743
9744 def trySynthStructInstance? (s : Struct) (expectedType : Expr) : TermElabM (Option Expr) := do
9745   if !s.allDefault then
9746     pure none
9747   else
9748     try synthInstance? expectedType catch _ => pure none
9749
9750 private partial def elabStruct (s : Struct) (expectedType? : Option Expr) : TermElabM (Expr × Struct) := withRef s.ref do
9751   let env ← getEnv
9752   let ctorVal := getStructureCtor env s.structName
9753   let { ctorFn := ctorFn, ctorFnType := ctorFnType, .. } ← mkCtorHeader ctorVal expectedType?
9754   let (e, _, fields) ← s.fields.foldlM (init := (ctorFn, ctorFnType, [])) fun (e, type, fields) field =>
9755     match field.lhs with
9756     | [FieldLHS.fieldName ref fieldName] => do
9757       let type ← whnfForall type
9758       match type with
9759       | Expr.forallE _ d b c =>
9760         let cont (val : Expr) (field : Field Struct) : TermElabM (Expr × Expr × Fields) := do
9761           pushInfoTree <| InfoTree.node (children := {}) <| Info.ofFieldInfo { lctx := (← getLCtx), val := val, name := fieldName, stx :=
9762             let e := mkApp e val
9763             let type := b.instantiate1 val
9764             let field := { field with expr? := some val }
9765             pure (e, type, field::fields)
9766         match field.val with
9767         | FieldVal.term stx => cont (← elabTermEnsuringType stx d) field
9768         | FieldVal.nested s => do
9769           -- if all fields of `s` are marked as `default`, then try to synthesize instance
9770           match (← trySynthStructInstance? s d) with
9771           | some val => cont val { field with val := FieldVal.term (mkHole field.ref) }
9772           | none => do let (val, sNew) ← elabStruct s (some d); let val ← ensureHasType d val; cont val { field with val := FieldVal
9773             | FieldVal.default => do let val ← withRef field.ref <| mkFreshExprMVar (some d); cont (markDefaultMissing val) field
9774             | _ => withRef field.ref <| throwFailedToElabField fieldName s.structName m!"unexpected constructor type{indentExpr type}"
9775             | _ => throwErrorAt field.ref "unexpected unexpanded structure field"

```



```

9776 pure (e, s.setFields fields.reverse)
9777
9778 namespace DefaultFields
9779
9780 structure Context where
9781   -- We must search for default values overridden in derived structures
9782   structs : Array Struct := #[]
9783   allStructNames : Array Name := #[]
9784   /--
9785   Consider the following example:
9786   ```
9787   structure A where
9788     x : Nat := 1
9789
9790   structure B extends A where
9791     y : Nat := x + 1
9792     x := y + 1
9793
9794   structure C extends B where
9795     z : Nat := 2*y
9796     x := z + 3
9797   ```
9798   And we are trying to elaborate a structure instance for `C`. There are default values for `x` at `A`, `B`, and `C`.
9799   We say the default value at `C` has distance 0, the one at `B` distance 1, and the one at `A` distance 2.
9800   The field `maxDistance` specifies the maximum distance considered in a round of Default field computation.
9801   Remark: since `C` does not set a default value of `y`, the default value at `B` is at distance 0.
9802
9803   The fixpoint for setting default values works in the following way.
9804   - Keep computing default values using `maxDistance == 0`.
9805   - We increase `maxDistance` whenever we failed to compute a new default value in a round.
9806   - If `maxDistance > 0`, then we interrupt a round as soon as we compute some default value.
9807     We use depth-first search.
9808   - We sign an error if no progress is made when `maxDistance` == structure hierarchy depth (2 in the example above).
9809   -/
9810   maxDistance : Nat := 0
9811
9812 structure State where
9813   progress : Bool := false
9814
9815 partial def collectStructNames (struct : Struct) (names : Array Name) : Array Name :=
9816   let names := names.push struct.structName
9817   struct.fields.foldl (init := names) fun names field =>
9818     match field.val with
9819     | FieldVal.nested struct => collectStructNames struct names
9820     | _ => names
9821
9822 partial def getHierarchyDepth (struct : Struct) : Nat :=

```

```

9823 struct.fields.foldl (init := 0) fun max field =>
9824   match field.val with
9825   | FieldVal.nested struct => Nat.max max (getHierarchyDepth struct + 1)
9826   | _ => max
9827
9828 partial def findDefaultMissing? (mctx : MetavarContext) (struct : Struct) : Option (Field Struct) :=
9829   struct.fields.findSome? fun field =>
9830     match field.val with
9831     | FieldVal.nested struct => findDefaultMissing? mctx struct
9832     | _ => match field.expr? with
9833     | none => unreachable!
9834     | some expr => match defaultMissing? expr with
9835     | some (Expr.mvar mvarId _) => if mctx.isExprAssigned mvarId then none else some field
9836     | _ => none
9837
9838 def getFieldName (field : Field Struct) : Name :=
9839   match field.lhs with
9840   | [FieldLHS.fieldName _ fieldName] => fieldName
9841   | _ => unreachable!
9842
9843 abbrev M := ReaderT Context (StateRefT State TermElabM)
9844
9845 def isRoundDone : M Bool := do
9846   return (← get).progress && (← read).maxDistance > 0
9847
9848 def getFieldValue? (struct : Struct) (fieldName : Name) : Option Expr :=
9849   struct.fields.findSome? fun field =>
9850     if getFieldName field == fieldName then
9851       field.expr?
9852     else
9853       none
9854
9855 partial def mkDefaultValueAux? (struct : Struct) : Expr → TermElabM (Option Expr)
9856 | Expr.lam n d b c => withRef struct.ref do
9857   if c.binderInfo.isExplicit then
9858     let fieldName := n
9859     match getFieldValue? struct fieldName with
9860     | none => pure none
9861     | some val =>
9862       let valType ← inferType val
9863       if (← isDefEq valType d) then
9864         mkDefaultValueAux? struct (b.instantiate1 val)
9865       else
9866         pure none
9867   else
9868     let arg ← mkFreshExprMVar d
9869     mkDefaultValueAux? struct (b.instantiate1 arg)

```

```

9870 | e =>
9871   if e.isAppOfArity `id 2 then
9872     pure (some e.appArg!)
9873   else
9874     pure (some e)
9875
9876 def mkDefaultValue? (struct : Struct) (cinfo : ConstantInfo) : TermElabM (Option Expr) :=
9877   withRef struct.ref do
9878     let us ← mkFreshLevelMVarsFor cinfo
9879     mkDefaultValueAux? struct (cinfo.instantiateValueLevelParams us)
9880
9881 /-- If `e` is a projection function of one of the given structures, then reduce it -/
9882 def reduceProjOf? (structNames : Array Name) (e : Expr) : MetaM (Option Expr) := do
9883   if !e.isApp then pure none
9884   else match e.getAppFn with
9885   | Expr.const name _ _ => do
9886     let env ← getEnv
9887     match env.getProjectionStructureName? name with
9888     | some structName =>
9889       if structNames.contains structName then
9890         Meta.unfoldDefinition? e
9891       else
9892         pure none
9893   | none => pure none
9894   | _ => pure none
9895
9896 /-- Reduce default value. It performs beta reduction and projections of the given structures. -/
9897 partial def reduce (structNames : Array Name) : Expr → MetaM Expr
9898 | e@(Expr.lam __ _ _) => lambdaLetTelescope e fun xs b => do mkLambdaFVars xs (← reduce structNames b)
9899 | e@(Expr.forallE _ _ _ _) => forallTelescope e fun xs b => do mkForallFVars xs (← reduce structNames b)
9900 | e@(Expr.letE _ _ _ _) => lambdaLetTelescope e fun xs b => do mkLetFVars xs (← reduce structNames b)
9901 | e@(Expr.proj _ i b _) => do
9902   match (← Meta.project? b i) with
9903   | some r => reduce structNames r
9904   | none => return e.updateProj! (← reduce structNames b)
9905 | e@(Expr.app f _ _) => do
9906   match (← reduceProjOf? structNames e) with
9907   | some r => reduce structNames r
9908   | none =>
9909     let f := f.getAppFn
9910     let f' ← reduce structNames f
9911     if f'.isLambda then
9912       let revArgs := e.getAppRevArgs
9913       reduce structNames (f'.betaRev revArgs)
9914     else
9915       let args ← e.getAppArgs.mapM (reduce structNames)
9916       return (mkAppN f' args)

```

```

9917 | e@(Expr.mdata _ b _) => do
9918   let b ← reduce structNames b
9919   if (defaultMissing? e).isSome && !b.isMVar then
9920     return b
9921   else
9922     return e.updateMData! b
9923 | e@(Expr.mvar mvarId _) => do
9924   match (← getExprMVarAssignment? mvarId) with
9925   | some val => if val.isMVar then reduce structNames val else pure val
9926   | none     => return e
9927 | e => return e
9928
9929 partial def tryToSynthesizeDefault (structs : Array Struct) (allStructNames : Array Name) (maxDistance : Nat) (fieldName : Name) (mvarId
9930   let rec loop (i : Nat) (dist : Nat) := do
9931     if dist > maxDistance then
9932       pure false
9933     else if h : i < structs.size then do
9934       let struct := structs.get (i, h)
9935       let defaultName := struct.structName ++ fieldName ++ `_default
9936       let env ← getEnv
9937       match env.find? defaultName with
9938       | some cinfo@(ConstantInfo.defnInfo defVal) => do
9939         let mctx ← getMCtx
9940         let val? ← mkDefaultValue? struct cinfo
9941         match val? with
9942         | none     => do setMCtx mctx; loop (i+1) (dist+1)
9943         | some val => do
9944           let val ← reduce allStructNames val
9945           match val.find? fun e => (defaultMissing? e).isSome with
9946           | some _ => setMCtx mctx; loop (i+1) (dist+1)
9947           | none     =>
9948             let mvarDecl ← getMVarDecl mvarId
9949             let val ← ensureHasType mvarDecl.type val
9950             assignExprMVar mvarId val
9951             pure true
9952         | _ => loop (i+1) dist
9953     else
9954       pure false
9955   loop 0 0
9956
9957 partial def step (struct : Struct) : M Unit :=
9958   unless (← isRoundDone) do
9959     withReader (fun ctx => { ctx with structs := ctx.structs.push struct }) do
9960       for field in struct.fields do
9961         match field.val with
9962         | FieldVal.nested struct => step struct
9963         | _ => match field.expr? with

```

```

9964 | none      => unreachable!
9965 | some expr => match defaultMissing? expr with
9966 | some (Expr.mvar mvarId _) =>
9967   unless (← isExprMVarAssigned mvarId) do
9968     let ctx ← read
9969     if (← withRef field.ref <| tryToSynthesizeDefault ctx.structs ctx.allStructNames ctx.maxDistance (getFieldName field) mv)
9970       modify fun s => { s with progress := true }
9971 | _ => pure ()
9972
9973 partial def propagateLoop (hierarchyDepth : Nat) (d : Nat) (struct : Struct) : M Unit := do
9974   match findDefaultMissing? (← getMCtx) struct with
9975   | none      => pure () -- Done
9976   | some field =>
9977     if d > hierarchyDepth then
9978       throwErrorAt field.ref "field '{getFieldName field}' is missing"
9979     else withReader (fun ctx => { ctx with maxDistance := d }) do
9980       modify fun s => { s with progress := false }
9981       step struct
9982       if (← get).progress then do
9983         propagateLoop hierarchyDepth 0 struct
9984       else
9985         propagateLoop hierarchyDepth (d+1) struct
9986
9987 def propagate (struct : Struct) : TermElabM Unit :=
9988   let hierarchyDepth := getHierarchyDepth struct
9989   let structNames := collectStructNames struct #[]
9990   (propagateLoop hierarchyDepth 0 struct { allStructNames := structNames }).run' {}
9991
9992 end DefaultFields
9993
9994 private def elabStructInstAux (stx : Syntax) (expectedType? : Option Expr) (source : Source) : TermElabM Expr := do
9995   let (structName, structType) ← getStructName stx expectedType? source
9996   unless isStructureLike (← getEnv) structName do
9997     throwError "invalid \{...\} notation, structure type expected{indentExpr structType}"
9998   match mkStructView stx structName source with
9999   | Except.error ex => throwError ex
10000 | Except.ok struct =>
10001   let struct ← expandStruct struct
10002   trace[Elab.struct] "{struct}"
10003   let (r, struct) ← elabStruct struct expectedType?
10004   DefaultFields.propagate struct
10005   pure r
10006
10007 @[builtinTermElab structInst] def elabStructInst : TermElab := fun stx expectedType? => do
10008   match (← expandNonAtomicExplicitSource stx) with
10009   | some stxNew => withMacroExpansion stx stxNew <| elabTerm stxNew expectedType?
10010   | none =>

```

```

10011   let sourceView ← getStructSource stx
10012   match (← isModifyOp? stx), sourceView with
10013   | some modifyOp, Source.explicit source _ => elabModifyOp stx modifyOp source expectedType?
10014   | some _,      _                        => throwError "invalid \{...\} notation, explicit source is required when using '[<index>]'"
10015   | _,          _                        => elabStructInstAux stx expectedType? sourceView
10016
10017 builtin_initialize registerTraceClass `Elab.struct
10018
10019 end Lean.Elab.Term.StructInst
10020 ::::::::::::::
10021 Elab/Structure.lean
10022 ::::::::::::::
10023 /-
10024 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
10025 Released under Apache 2.0 license as described in the file LICENSE.
10026 Authors: Leonardo de Moura
10027 -/
10028 import Lean.Parser.Command
10029 import Lean.Meta.Closure
10030 import Lean.Meta.SizeOf
10031 import Lean.Elab.Command
10032 import Lean.Elab.DeclModifiers
10033 import Lean.Elab.DeclUtil
10034 import Lean.Elab.Inductive
10035 import Lean.Elab.DeclarationRange
10036
10037 namespace Lean.Elab.Command
10038
10039 open Meta
10040
10041 /- Recall that the `structure` command syntax is
10042 ```
10043 leading_parser (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType >> optional (" := "
10044 ```
10045 -/
10046
10047 structure StructCtorView where
10048   ref      : Syntax
10049   modifiers : Modifiers
10050   inferMod  : Bool -- true if `{}` is used in the constructor declaration
10051   name      : Name
10052   declName  : Name
10053
10054 structure StructFieldView where
10055   ref      : Syntax
10056   modifiers : Modifiers
10057   binderInfo : BinderInfo

```

```

10058 inferMod    : Bool
10059 declName     : Name
10060 name          : Name
10061 binders       : Syntax
10062 type?         : Option Syntax
10063 value?        : Option Syntax
10064
10065 structure StructView where
10066   ref          : Syntax
10067   modifiers     : Modifiers
10068   scopeLevelNames : List Name -- All `universe` declarations in the current scope
10069   allUserLevelNames : List Name -- `scopeLevelNames` ++ explicit universe parameters provided in the `structure` command
10070   isClass       : Bool
10071   declName      : Name
10072   scopeVars     : Array Expr -- All `variable` declaration in the current scope
10073   params        : Array Expr -- Explicit parameters provided in the `structure` command
10074   parents       : Array Syntax
10075   type          : Syntax
10076   ctor          : StructCtorView
10077   fields        : Array StructFieldView
10078
10079 inductive StructFieldKind where
10080   | newField | fromParent | subobject
10081   deriving Inhabited
10082
10083 structure StructFieldInfo where
10084   name      : Name
10085   declName  : Name -- Remark: this field value doesn't matter for fromParent fields.
10086   fvar      : Expr
10087   kind      : StructFieldKind
10088   inferMod  : Bool := false
10089   value?    : Option Expr := none
10090   deriving Inhabited
10091
10092 def StructFieldInfo.isFromParent (info : StructFieldInfo) : Bool :=
10093   match info.kind with
10094   | StructFieldKind.fromParent => true
10095   | _                          => false
10096
10097 def StructFieldInfo.isSubobject (info : StructFieldInfo) : Bool :=
10098   match info.kind with
10099   | StructFieldKind.subobject => true
10100   | _                        => false
10101
10102 /- Auxiliary declaration for `mkProjections` -/
10103 structure ProjectionInfo where
10104   declName : Name

```

```

10105 inferMod : Bool
10106
10107 structure ElabStructResult where
10108   decl          : Declaration
10109   projInfos      : List ProjectionInfo
10110   projInstances  : List Name -- projections (to parent classes) that must be marked as instances.
10111   mctx           : MetavarContext
10112   lctx           : LocalContext
10113   localInsts     : LocalInstances
10114   defaultAuxDecls : Array (Name × Expr × Expr)
10115
10116 private def defaultCtorName := `mk
10117
10118 /-
10119 The structure constructor syntax is
10120 ```
10121 leading_parser try (declModifiers >> ident >> optional inferMod >> " :: ")
10122 ```
10123 -/
10124 private def expandCtor (structStx : Syntax) (structModifiers : Modifiers) (structDeclName : Name) : TermElabM StructCtorView := do
10125   let useDefault := do
10126     let declName := structDeclName ++ defaultCtorName
10127     addAuxDeclarationRanges declName structStx[2] structStx[2]
10128     pure { ref := structStx, modifiers := {}, inferMod := false, name := defaultCtorName, declName := declName }
10129   if structStx[5].isNone then
10130     useDefault
10131   else
10132     let optCtor := structStx[5][1]
10133     if optCtor.isNone then
10134       useDefault
10135     else
10136       let ctor := optCtor[0]
10137       withRef ctor do
10138         let ctorModifiers ← elabModifiers ctor[0]
10139         checkValidCtorModifier ctorModifiers
10140         if ctorModifiers.isPrivate && structModifiers.isPrivate then
10141           throwError "invalid 'private' constructor in a 'private' structure"
10142         if ctorModifiers.isProtected && structModifiers.isPrivate then
10143           throwError "invalid 'protected' constructor in a 'private' structure"
10144         let inferMod := !ctor[2].isNone
10145         let name := ctor[1].getId
10146         let declName := structDeclName ++ name
10147         let declName ← applyVisibility ctorModifiers.visibility declName
10148         addDocString' declName ctorModifiers.docString?
10149         addAuxDeclarationRanges declName ctor[1] ctor[1]
10150         pure { ref := ctor, name := name, modifiers := ctorModifiers, inferMod := inferMod, declName := declName }
10151

```



```

10152 def checkValidFieldModifier (modifiers : Modifiers) : TermElabM Unit := do
10153   if modifiers.isNoncomputable then
10154     throwError "invalid use of 'noncomputable' in field declaration"
10155   if modifiers.isPartial then
10156     throwError "invalid use of 'partial' in field declaration"
10157   if modifiers.isUnsafe then
10158     throwError "invalid use of 'unsafe' in field declaration"
10159   if modifiers.attrs.size != 0 then
10160     throwError "invalid use of attributes in field declaration"
10161   if modifiers.isPrivate then
10162     throwError "private fields are not supported yet"
10163
10164   /-
10165   ```
10166   def structExplicitBinder := leading_parser atomic (declModifiers true >> "(" >> many1 ident >> optional inferMod >> optDeclSig >> opti
10167   def structImplicitBinder := leading_parser atomic (declModifiers true >> "{" >> many1 ident >> optional inferMod >> declSig >> "}"
10168   def structInstBinder := leading_parser atomic (declModifiers true >> "[" >> many1 ident >> optional inferMod >> declSig >> "]"
10169   def structSimpleBinder := leading_parser atomic (declModifiers true >> ident) >> optional inferMod >> optDeclSig >> optional Term.bin
10170   def structFields := leading_parser many (structExplicitBinder <|> structImplicitBinder <|> structInstBinder)
10171   ```
10172   -/
10173 private def expandFields (structStx : Syntax) (structModifiers : Modifiers) (structDeclName : Name) : TermElabM (Array StructFieldView)
10174   let fieldBinders := if structStx[5].isNone then #[] else structStx[5][2][0].getArgs
10175   fieldBinders.foldlM (init := #[]) fun (views : Array StructFieldView) fieldBinder => withRef fieldBinder do
10176     let mut fieldBinder := fieldBinder
10177     if fieldBinder.getKind == ``Parser.Command.structSimpleBinder then
10178       fieldBinder := Syntax.node ``Parser.Command.structExplicitBinder
10179       #[ fieldBinder[0], mkAtomFrom fieldBinder "(", mkNullNode #[ fieldBinder[1] ], fieldBinder[2], fieldBinder[3], fieldBinder[4],
10180     let k := fieldBinder.getKind
10181     let binfo ←
10182       if k == ``Parser.Command.structExplicitBinder then pure BinderInfo.default
10183       else if k == ``Parser.Command.structImplicitBinder then pure BinderInfo.implicit
10184       else if k == ``Parser.Command.structInstBinder then pure BinderInfo.instImplicit
10185       else throwError "unexpected kind of structure field"
10186     let fieldModifiers ← elabModifiers fieldBinder[0]
10187     checkValidFieldModifier fieldModifiers
10188     if fieldModifiers.isPrivate && structModifiers.isPrivate then
10189       throwError "invalid 'private' field in a 'private' structure"
10190     if fieldModifiers.isProtected && structModifiers.isPrivate then
10191       throwError "invalid 'protected' field in a 'private' structure"
10192     let inferMod := !fieldBinder[3].isNone
10193     let (binders, type?) :=
10194       if binfo == BinderInfo.default then
10195         expandOptDeclSig fieldBinder[4]
10196       else
10197         let (binders, type) := expandDeclSig fieldBinder[4]
10198         (binders, some type)

```

```

10199   let value? :=
10200     if binfo != BinderInfo.default then none
10201     else
10202       let optBinderDefault := fieldBinder[5]
10203       if optBinderDefault.isNone then none
10204       else
10205         -- binderDefault := leading_parser " := " >> termParser
10206         some optBinderDefault[0][1]
10207   let ident := fieldBinder[2].getArgs
10208   idents.foldlM (init := views) fun (views : Array StructFieldView) ident => withRef ident do
10209     let name      := ident.getId
10210     if isInternalSubobjectFieldName name then
10211       throwError "invalid field name '{name}', identifiers starting with '_' are reserved to the system"
10212     let declName := structDeclName ++ name
10213     let declName ← applyVisibility fieldModifiers.visibility declName
10214     addDocString' declName fieldModifiers.docString?
10215     return views.push {
10216       ref      := ident,
10217       modifiers := fieldModifiers,
10218       binderInfo := binfo,
10219       inferMod   := inferMod,
10220       declName   := declName,
10221       name       := name,
10222       binders    := binders,
10223       type?      := type?,
10224       value?     := value?
10225     }
10226
10227 private def validStructType (type : Expr) : Bool :=
10228   match type with
10229   | Expr.sort .. => true
10230   | _            => false
10231
10232 private def checkParentIsStructure (parent : Expr) : TermElabM Name :=
10233   match parent.getAppFn with
10234   | Expr.const c _ => do
10235     unless isStructure (← getEnv) c do
10236       throwError "'{c}' is not a structure"
10237     pure c
10238   | _ => throwError "expected structure"
10239
10240 private def findFieldInfo? (infos : Array StructFieldInfo) (fieldName : Name) : Option StructFieldInfo :=
10241   infos.find? fun info => info.name == fieldName
10242
10243 private def containsFieldName (infos : Array StructFieldInfo) (fieldName : Name) : Bool :=
10244   (findFieldInfo? infos fieldName).isSome
10245

```

```

10246 private partial def processSubfields (structDeclName : Name) (parentFVar : Expr) (parentStructName : Name) (subfieldNames : Array Name)
10247   (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α) : TermElabM α :=
10248   let rec loop (i : Nat) (infos : Array StructFieldInfo) := do
10249     if h : i < subfieldNames.size then
10250       let subfieldName := subfieldNames.get {i, h}
10251       if containsFieldName infos subfieldName then
10252         throwError "field '{subfieldName}' from '{parentStructName}' has already been declared"
10253       let val ← mkProjection parentFVar subfieldName
10254       let type ← inferType val
10255       withLetDecl subfieldName type val fun subfieldFVar =>
10256         /- The following `declName` is only used for creating the `_default` auxiliary declaration name when
10257         its default value is overwritten in the structure. -/
10258         let declName := structDeclName ++ subfieldName
10259         let infos := infos.push { name := subfieldName, declName := declName, fvar := subfieldFVar, kind := StructFieldKind.fromParent
10260         loop (i+1) infos
10261     else
10262       k infos
10263   loop 0 infos
10264
10265 private partial def withParents (view : StructView) (i : Nat) (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α)
10266   if h : i < view.parents.size then
10267     let parentStx := view.parents.get {i, h}
10268     withRef parentStx do
10269       let parent ← Term.elabType parentStx
10270       let parentName ← checkParentIsStructure parent
10271       let toParentName := Name.mkSimple $ "to" ++ parentName.eraseMacroScopes.getString! -- erase macro scopes?
10272       if containsFieldName infos toParentName then
10273         throwErrorAt parentStx "field '{toParentName}' has already been declared"
10274       let env ← getEnv
10275       let binfo := if view.isClass && isClass env parentName then BinderInfo.instImplicit else BinderInfo.default
10276       withLocalDecl toParentName binfo parent fun parentFVar =>
10277         let infos := infos.push { name := toParentName, declName := view.declName ++ toParentName, fvar := parentFVar, kind := StructFieldKind.fromParent
10278         let subfieldNames := getStructureFieldsFlattened env parentName
10279         processSubfields view.declName parentFVar parentName subfieldNames infos fun infos => withParents view (i+1) infos k
10280     else
10281       k infos
10282
10283 private def elabFieldTypeValue (view : StructFieldView) : TermElabM (Option Expr × Option Expr) := do
10284   Term.withAutoBoundImplicit <| Term.elabBinders view.binders.getArgs fun params => do
10285     match view.type? with
10286     | none =>
10287       match view.value? with
10288       | none => return (none, none)
10289       | some valStx =>
10290         Term.synthesizeSyntheticMVarsNoPostponing
10291         let params ← Term.addAutoBoundImplicits params
10292         let value ← Term.elabTerm valStx none

```

```

10293     let value ← mkLambdaFVars params value
10294     return (none, value)
10295 | some typeStx =>
10296   let type ← Term.elabType typeStx
10297   Term.synthesizeSyntheticMVarsNoPostponing
10298   let params ← Term.addAutoBoundImplicits params
10299   match view.value? with
10300   | none =>
10301     let type ← mkForallFVars params type
10302     return (type, none)
10303   | some valStx =>
10304     let value ← Term.elabTermEnsuringType valStx type
10305     Term.synthesizeSyntheticMVarsNoPostponing
10306     let type ← mkForallFVars params type
10307     let value ← mkLambdaFVars params value
10308     return (type, value)
10309
10310 private partial def withFields
10311   (views : Array StructFieldView) (i : Nat) (infos : Array StructFieldInfo) (k : Array StructFieldInfo → TermElabM α) : TermElabM α :=
10312   if h : i < views.size then
10313     let view := views.get (i, h)
10314     withRef view.ref $
10315     match findFieldInfo? infos view.name with
10316     | none => do
10317       let (type?, value?) ← elabFieldTypeValue view
10318       match type?, value? with
10319       | none, none => throwError "invalid field, type expected"
10320       | some type, _ =>
10321         withLocalDecl view.name view.binderInfo type fun fieldFVar =>
10322           let infos := infos.push { name := view.name, declName := view.declName, fvar := fieldFVar, value? := value?,
10323             kind := StructFieldKind.newField, inferMod := view.inferMod }
10324           withFields views (i+1) infos k
10325     | none, some value =>
10326       let type ← inferType value
10327       withLocalDecl view.name view.binderInfo type fun fieldFVar =>
10328         let infos := infos.push { name := view.name, declName := view.declName, fvar := fieldFVar, value? := value,
10329           kind := StructFieldKind.newField, inferMod := view.inferMod }
10330         withFields views (i+1) infos k
10331   | some info =>
10332     match info.kind with
10333     | StructFieldKind.newField => throwError "field '{view.name}' has already been declared"
10334     | StructFieldKind.fromParent =>
10335       match view.value? with
10336       | none => throwError "field '{view.name}' has been declared in parent structure"
10337       | some valStx => do
10338         if let some type := view.type? then
10339           throwErrorAt type "omit field '{view.name}' type to set default value"

```

```

10340         else
10341             let mut valStx := valStx
10342             if view.binders.getArgs.size > 0 then
10343                 valStx ← `(fun $(view.binders.getArgs)* => $valStx:term)
10344                 let fvarType ← inferType info.fvar
10345                 let value ← Term.elabTermEnsuringType valStx fvarType
10346                 let infos := infos.push { info with value? := value }
10347                 withFields views (i+1) infos k
10348             | StructFieldKind.subobject => unreachable!
10349         else
10350             k infos
10351
10352 private def getResultUniverse (type : Expr) : TermElabM Level := do
10353     let type ← whnf type
10354     match type with
10355     | Expr.sort u _ => pure u
10356     | _             => throwError "unexpected structure resulting type"
10357
10358 private def collectUsed (params : Array Expr) (fieldInfos : Array StructFieldInfo) : StateRefT CollectFVars.State MetaM Unit := do
10359     params.forM fun p => do
10360         let type ← inferType p
10361         Term.collectUsedFVars type
10362     fieldInfos.forM fun info => do
10363         let fvarType ← inferType info.fvar
10364         Term.collectUsedFVars fvarType
10365         match info.value? with
10366         | none      => pure ()
10367         | some value => Term.collectUsedFVars value
10368
10369 private def removeUnused (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo)
10370     : TermElabM (LocalContext × LocalInstances × Array Expr) := do
10371     let (_, used) ← (collectUsed params fieldInfos).run {}
10372     Term.removeUnused scopeVars used
10373
10374 private def withUsed {α} (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo) (k : Array Expr → TermElabM
10375     : TermElabM α := do
10376     let (lctx, localInsts, vars) ← removeUnused scopeVars params fieldInfos
10377     withLCtx lctx localInsts $ k vars
10378
10379 private def levelMVarToParamFVar (fvar : Expr) : StateRefT Nat TermElabM Unit := do
10380     let type ← inferType fvar
10381     discard <| Term.levelMVarToParam' type
10382
10383 private def levelMVarToParamFVars (fvars : Array Expr) : StateRefT Nat TermElabM Unit :=
10384     fvars.forM levelMVarToParamFVar
10385
10386 private def levelMVarToParamAux (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo)

```

```

10387 : StateRefT Nat TermElabM (Array StructFieldInfo) := do
10388 levelMVarToParamFVars scopeVars
10389 levelMVarToParamFVars params
10390 fieldInfos.mapM fun info => do
10391   levelMVarToParamFVar info.fvar
10392   match info.value? with
10393   | none      => pure info
10394   | some value =>
10395     let value ← Term.levelMVarToParam' value
10396     pure { info with value? := value }
10397
10398 private def levelMVarToParam (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array StructFieldInfo) : TermElabM (Array Str
10399   (levelMVarToParamAux scopeVars params fieldInfos).run' 1
10400
10401 private partial def collectUniversesFromFields (r : Level) (rOffset : Nat) (fieldInfos : Array StructFieldInfo) : TermElabM (Array Level) := do
10402   fieldInfos.foldlM (init := #[]) fun (us : Array Level) (info : StructFieldInfo) => do
10403     let type ← inferType info.fvar
10404     let u ← getLevel type
10405     let u ← instantiateLevelMVars u
10406     accLevelAtCtor u r rOffset us
10407
10408 private def updateResultingUniverse (fieldInfos : Array StructFieldInfo) (type : Expr) : TermElabM Expr := do
10409   let r ← getResultUniverse type
10410   let rOffset : Nat := r.getOffset
10411   let r : Level := r.getLevelOffset
10412   match r with
10413   | Level.mvar mvarId _ =>
10414     let us ← collectUniversesFromFields r rOffset fieldInfos
10415     let rNew := mkResultUniverse us rOffset
10416     assignLevelMVar mvarId rNew
10417     instantiateMVars type
10418   | _ => throwError "failed to compute resulting universe level of structure, provide universe explicitly"
10419
10420 private def collectLevelParamsInFVar (s : CollectLevelParams.State) (fvar : Expr) : TermElabM CollectLevelParams.State := do
10421   let type ← inferType fvar
10422   let type ← instantiateMVars type
10423   pure $ collectLevelParams s type
10424
10425 private def collectLevelParamsInFVars (fvvars : Array Expr) (s : CollectLevelParams.State) : TermElabM CollectLevelParams.State :=
10426   fvvars.foldlM collectLevelParamsInFVar s
10427
10428 private def collectLevelParamsInStructure (structType : Expr) (scopeVars : Array Expr) (params : Array Expr) (fieldInfos : Array Struct
10429   : TermElabM (Array Name) := do
10430   let s := collectLevelParams {} structType
10431   let s ← collectLevelParamsInFVars scopeVars s
10432   let s ← collectLevelParamsInFVars params s
10433   let s ← fieldInfos.foldlM (fun (s : CollectLevelParams.State) info => collectLevelParamsInFVar s info.fvar) s

```

```

10434 pure s.params
10435
10436 private def addCtorFields (fieldInfos : Array StructFieldInfo) : Nat → Expr → TermElabM Expr
10437 | 0, type => pure type
10438 | i+1, type => do
10439   let info := fieldInfos[i]
10440   let decl ← Term.getFVarLocalDecl! info.fvar
10441   let type ← instantiateMVars type
10442   let type := type.abstract #[info.fvar]
10443   match info.kind with
10444   | StructFieldKind.fromParent =>
10445     let val := decl.value
10446     addCtorFields fieldInfos i (type.instantiate1 val)
10447   | StructFieldKind.subobject =>
10448     let n := mkInternalSubobjectFieldName $ decl.userName
10449     addCtorFields fieldInfos i (mkForall n decl.binderInfo decl.type type)
10450   | StructFieldKind.newField =>
10451     addCtorFields fieldInfos i (mkForall decl.userName decl.binderInfo decl.type type)
10452
10453 private def mkCtor (view : StructView) (levelParams : List Name) (params : Array Expr) (fieldInfos : Array StructFieldInfo) : TermElabM
10454 withRef view.ref do
10455   let type := mkAppN (mkConst view.declName (levelParams.map mkLevelParam)) params
10456   let type ← addCtorFields fieldInfos fieldInfos.size type
10457   let type ← mkForallFVars params type
10458   let type ← instantiateMVars type
10459   let type := type.inferImplicit params.size !view.ctor.inferMod
10460   pure { name := view.ctor.declName, type := type }
10461
10462 @[extern "lean_mk_projections"]
10463 private constant mkProjections (env : Environment) (structName : Name) (projs : List ProjectionInfo) (isClass : Bool) : Except KernelEx
10464
10465 private def addProjections (structName : Name) (projs : List ProjectionInfo) (isClass : Bool) : TermElabM Unit := do
10466   let env ← getEnv
10467   match mkProjections env structName projs isClass with
10468   | Except.ok env => setEnv env
10469   | Except.error ex => throwKernelException ex
10470
10471 private def mkAuxConstructions (declName : Name) : TermElabM Unit := do
10472   let env ← getEnv
10473   let hasUnit := env.contains `PUnit
10474   let hasEq := env.contains `Eq
10475   let hasHEq := env.contains `HEq
10476   mkRecOn declName
10477   if hasUnit then mkCasesOn declName
10478   if hasUnit && hasEq && hasHEq then mkNoConfusion declName
10479
10480 private def addDefaults (lctx : LocalContext) (defaultAuxDecls : Array (Name × Expr × Expr)) : TermElabM Unit := do

```



```

10481 let localInsts ← getLocalInstances
10482 withLCtx lctx localInsts do
10483   defaultAuxDecls.forM fun (declName, type, value) => do
10484     let value ← instantiateMVars value
10485     if value.hasExprMVar then
10486       throwError "invalid default value for field, it contains metavariables{indentExpr value}"
10487     /- The identity function is used as "marker". -/
10488     let value ← mkId value
10489     discard <| mkAuxDefinition declName type value (zeta := true)
10490     setReducibleAttribute declName
10491
10492 private def elabStructureView (view : StructView) : TermElabM Unit := do
10493   view.fields.forM fun field => do
10494     if field.declName == view.ctor.declName then
10495       throwErrorAt field.ref "invalid field name '{field.name}', it is equal to structure constructor name"
10496     addAuxDeclarationRanges field.declName field.ref field.ref
10497   let numExplicitParams := view.params.size
10498   let type ← Term.elabType view.type
10499   unless validStructType type do throwErrorAt view.type "expected Type"
10500   withRef view.ref do
10501     withParents view 0 #[] fun fieldInfos =>
10502       withFields view.fields 0 fieldInfos fun fieldInfos => do
10503         Term.synthesizeSyntheticMVarsNoPostponing
10504         let u ← getResultUniverse type
10505         let inferLevel ← shouldInferResultUniverse u
10506         withUsed view.scopeVars view.params fieldInfos $ fun scopeVars => do
10507           let numParams := scopeVars.size + numExplicitParams
10508           let fieldInfos ← levelMVarToParam scopeVars view.params fieldInfos
10509           let type ← withRef view.ref do
10510             if inferLevel then
10511               updateResultingUniverse fieldInfos type
10512             else
10513               checkResultingUniverse (← getResultUniverse type)
10514               pure type
10515           trace[Elab.structure] "type: {type}"
10516           let usedLevelNames ← collectLevelParamsInStructure type scopeVars view.params fieldInfos
10517           match sortDeclLevelParams view.scopeLevelNames view.allUserLevelNames usedLevelNames with
10518           | Except.error msg      => withRef view.ref <| throwError msg
10519           | Except.ok levelParams =>
10520             let params := scopeVars ++ view.params
10521             let ctor ← mkCtor view levelParams params fieldInfos
10522             let type ← mkForallFVars params type
10523             let type ← instantiateMVars type
10524             let indType := { name := view.declName, type := type, ctors := [ctor] : InductiveType }
10525             let decl := Declaration.inductDecl levelParams params.size [indType] view.modifiers.isUnsafe
10526             Term.ensureNoUnassignedMVars decl
10527             addDecl decl

```



```

10528 let projInfos := (fieldInfos.filter fun (info : StructFieldInfo) => !info.isFromParent).toList.map fun (info : StructFieldInfo)
10529   { declName := info.declName, inferMod := info.inferMod : ProjectionInfo }
10530 addProjections view.declName projInfos view.isClass
10531 mkAuxConstructions view.declName
10532 let instParents ← fieldInfos.filterM fun info => do
10533   let decl ← Term.getFVarLocalDecl! info.fvar
10534   pure (info.isSubobject && decl.binderInfo.isInstImplicit)
10535 let projInstances := instParents.toList.map fun info => info.declName
10536 Term.applyAttributesAt view.declName view.modifiers.attrs AttributeApplicationTime.afterTypeChecking
10537 projInstances.forM fun declName => addInstance declName AttributeKind.global (eval_prio default)
10538 let lctx ← getLCtx
10539 let fieldsWithDefault := fieldInfos.filter fun info => info.value?.isSome
10540 let defaultAuxDecls ← fieldsWithDefault.mapM fun info => do
10541   let type ← inferType info.fvar
10542   pure (info.declName ++ `_default`, type, info.value?.get!)
10543 /- The `lctx` and `defaultAuxDecls` are used to create the auxiliary `_default` declarations
10544 The parameters `params` for these definitions must be marked as implicit, and all others as explicit. -/
10545 let lctx :=
10546   params.foldl (init := lctx) fun (lctx : LocalContext) (p : Expr) =>
10547     lctx.setBinderInfo p.fvarId! BinderInfo.implicit
10548 let lctx :=
10549   fieldInfos.foldl (init := lctx) fun (lctx : LocalContext) (info : StructFieldInfo) =>
10550     if info.isFromParent then lctx -- `fromParent` fields are elaborated as let-decls, and are zeta-expanded when creating `_de
10551     else lctx.setBinderInfo info.fvar.fvarId! BinderInfo.default
10552 addDefaults lctx defaultAuxDecls
10553
10554 /-
10555 leading_parser (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType >> " := " >> opti
10556
10557 where
10558 def «extends» := leading_parser " extends " >> sepBy1 termParser ", "
10559 def typeSpec := leading_parser " : " >> termParser
10560 def optType : Parser := optional typeSpec
10561
10562 def structFields := leading_parser many (structExplicitBinder <|> structImplicitBinder <|> structInstBinder)
10563 def structCtor := leading_parser try (declModifiers >> ident >> optional inferMod >> " :: ")
10564
10565 -/
10566 def elabStructure (modifiers : Modifiers) (stx : Syntax) : CommandElabM Unit := do
10567   checkValidInductiveModifier modifiers
10568   let isClass := stx[0].getKind == `Parser.Command.classTk
10569   let modifiers := if isClass then modifiers.addAttribute { name := `class } else modifiers
10570   let declId := stx[1]
10571   let params := stx[2].getArgs
10572   let exts := stx[3]
10573   let parents := if exts.isNone then #[] else exts[0][1].getSepArgs
10574   let optType := stx[4]

```

```

10575 let derivingClassViews ← getOptDerivingClasses stx[6]
10576 let type ← if optType.isNone then `(Sort _) else pure optType[0][1]
10577 let declName ←
10578   runTermElabM none fun scopeVars => do
10579     let scopeLevelNames ← Term.getLevelNames
10580     let (name, declName, allUserLevelNames) ← Elab.expandDeclId (← getCurrNamespace) scopeLevelNames declId modifiers
10581     addDeclarationRanges declName stx
10582     Term.withDeclName declName do
10583       let ctor ← expandCtor stx modifiers declName
10584       let fields ← expandFields stx modifiers declName
10585       Term.withLevelNames allUserLevelNames <| Term.withAutoBoundImplicit <|
10586         Term.elabBinders params fun params => do
10587           Term.synthesizeSyntheticMVarsNoPostponing
10588           let params ← Term.addAutoBoundImplicits params
10589           let allUserLevelNames ← Term.getLevelNames
10590           elabStructureView {
10591             ref          := stx
10592             modifiers    := modifiers
10593             scopeLevelNames := scopeLevelNames
10594             allUserLevelNames := allUserLevelNames
10595             declName     := declName
10596             isClass      := isClass
10597             scopeVars    := scopeVars
10598             params       := params
10599             parents      := parents
10600             type         := type
10601             ctor         := ctor
10602             fields       := fields
10603           }
10604           unless isClass do
10605             mkSizeOfInstances declName
10606           return declName
10607   derivingClassViews.forM fun view => view.applyHandlers #[declName]
10608
10609 builtin_initialize registerTraceClass `Elab.structure
10610
10611 end Lean.Elab.Command
10612 :::::::::::::::
10613 Elab/Syntax.lean
10614 :::::::::::::::
10615 /-
10616 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
10617 Released under Apache 2.0 license as described in the file LICENSE.
10618 Authors: Leonardo de Moura
10619 -/
10620 import Lean.Elab.Command
10621 import Lean.Parser.Syntax

```

```

10622
10623 namespace Lean.Elab.Term
10624 /-
10625 Expand `optional «precedence»` where
10626 «precedence» := leading_parser " : " >> precedenceParser -/
10627 def expandOptPrecedence (stx : Syntax) : MacroM (Option Nat) :=
10628   if stx.isNone then
10629     return none
10630   else
10631     return some (← evalPrec stx[0][1])
10632
10633 private def mkParserSeq (ds : Array Syntax) : TermElabM Syntax := do
10634   if ds.size == 0 then
10635     throwUnsupportedSyntax
10636   else if ds.size == 1 then
10637     pure ds[0]
10638   else
10639     let mut r := ds[0]
10640     for d in ds[1:ds.size] do
10641       r ← `(ParserDescr.binary `andthen $r $d)
10642     return r
10643
10644 structure ToParserDescrContext where
10645   catName   : Name
10646   first     : Bool
10647   leftRec   : Bool -- true iff left recursion is allowed
10648   /- See comment at `Parser.ParserCategory`. -/
10649   behavior  : Parser.LeaningIdentBehavior
10650
10651 abbrev ToParserDescrM := ReaderT ToParserDescrContext (StateRefT (Option Nat) TermElabM)
10652 private def markAsTrailingParser (lhsPrec : Nat) : ToParserDescrM Unit := set (some lhsPrec)
10653
10654 @[inline] private def withNotFirst {α} (x : ToParserDescrM α) : ToParserDescrM α :=
10655   withReader (fun ctx => { ctx with first := false }) x
10656
10657 @[inline] private def withNestedParser {α} (x : ToParserDescrM α) : ToParserDescrM α :=
10658   withReader (fun ctx => { ctx with leftRec := false, first := false }) x
10659
10660 def checkLeftRec (stx : Syntax) : ToParserDescrM Bool := do
10661   let ctx ← read
10662   unless ctx.first && stx.getKind == `Lean.Parser.Syntax.cat do
10663     return false
10664   let cat := stx[0].getId.eraseMacroScopes
10665   unless cat == ctx.catName do
10666     return false
10667   let prec? ← liftMacroM <| expandOptPrecedence stx[1]
10668   unless ctx.leftRec do

```

```

10669     throwErrorAt stx[3] "invalid occurrence of '{cat}', parser algorithm does not allow this form of left recursion"
10670   markAsTrailingParser (prec?.getD 0)
10671   return true
10672
10673 /--
10674   Given a `stx` of category `syntax`, return a pair `(newStx, lhsPrec?)`,
10675   where `newStx` is of category `term`. After elaboration, `newStx` should have type
10676   `TrailingParserDescr` if `lhsPrec?.isSome`, and `ParserDescr` otherwise. -/
10677 partial def toParserDescr (stx : Syntax) (catName : Name) : TermElabM (Syntax × Option Nat) := do
10678   let env ← getEnv
10679   let behavior := Parser.leadingIdentBehavior env catName
10680   (process stx { catName := catName, first := true, leftRec := true, behavior := behavior }).run none
10681 where
10682   process (stx : Syntax) : ToParserDescrM Syntax := withRef stx do
10683     let kind := stx.getKind
10684     if kind == nullKind then
10685       processSeq stx
10686     else if kind == choiceKind then
10687       process stx[0]
10688     else if kind == `Lean.Parser.Syntax.paren then
10689       process stx[1]
10690     else if kind == `Lean.Parser.Syntax.cat then
10691       processNullaryOrCat stx
10692     else if kind == `Lean.Parser.Syntax.unary then
10693       processUnary stx
10694     else if kind == `Lean.Parser.Syntax.binary then
10695       processBinary stx
10696     else if kind == `Lean.Parser.Syntax.sepBy then
10697       processSepBy stx
10698     else if kind == `Lean.Parser.Syntax.sepBy1 then
10699       processSepBy1 stx
10700     else if kind == `Lean.Parser.Syntax.atom then
10701       processAtom stx
10702     else if kind == `Lean.Parser.Syntax.nonReserved then
10703       processNonReserved stx
10704     else
10705       let stxNew? ← liftM (liftMacroM (expandMacro? stx) : TermElabM _)
10706       match stxNew? with
10707       | some stxNew => process stxNew
10708       | none => throwErrorAt stx "unexpected syntax kind of category `syntax`: {kind}"
10709
10710 /- Sequence (aka NullNode) -/
10711 processSeq (stx : Syntax) := do
10712   let args := stx.getArgs
10713   if (← checkLeftRec stx[0]) then
10714     if args.size == 1 then throwErrorAt stx "invalid atomic left recursive syntax"
10715     let args := args.eraseIdx 0

```

```

10716     let args ← args.mapM fun arg => withNestedParser do process arg
10717     mkParserSeq args
10718   else
10719     let args ← args.mapIdxM fun i arg => withReader (fun ctx => { ctx with first := ctx.first && i.val == 0 }) do process arg
10720     mkParserSeq args
10721
10722   /- Resolve the given parser name and return a list of candidates.
10723   Each candidate is a pair `(resolvedParserName, isDescr)`.
10724   `isDescr == true` if the type of `resolvedParserName` is a `ParserDescr`. -/
10725   resolveParserName (parserName : Name) : ToParserDescrM (List (Name × Bool)) := do
10726     try
10727       let candidates ← resolveGlobalConstWithInfos (← getRef) parserName
10728       /- Convert `candidates` in a list of pairs `(c, isDescr)`, where `c` is the parser name,
10729       and `isDescr` is true iff `c` has type `Lean.ParserDescr` or `Lean.TrailingParser` -/
10730       let env ← getEnv
10731       candidates.filterMap fun c =>
10732         match env.find? c with
10733         | none      => none
10734         | some info =>
10735           match info.type with
10736           | Expr.const `Lean.Parser.TrailingParser _ _ => (c, false)
10737           | Expr.const `Lean.Parser.Parser _ _         => (c, false)
10738           | Expr.const `Lean.ParserDescr _ _           => (c, true)
10739           | Expr.const `Lean.TrailingParserDescr _ _   => (c, true)
10740           | _                                           => none
10741       catch _ => return []
10742
10743   ensureNoPrec (stx : Syntax) :=
10744     unless stx[1].isNone do
10745       throwErrorAt stx[1] "unexpected precedence"
10746
10747   processParserCategory (stx : Syntax) := do
10748     let catName := stx[0].getId.eraseMacroScopes
10749     if (← read).first && catName == (← read).catName then
10750       throwErrorAt stx "invalid atomic left recursive syntax"
10751     let prec? ← liftMacroM <| expandOptPrecedence stx[1]
10752     let prec := prec?.getD 0
10753     `(ParserDescr.cat $(quote catName) $(quote prec))
10754
10755   processNullaryOrCat (stx : Syntax) := do
10756     let id := stx[0].getId.eraseMacroScopes
10757     match (← withRef stx[0] <| resolveParserName id) with
10758     | [(c, true)]      => ensureNoPrec stx; return mkIdentFrom stx c
10759     | [(c, false)]    => ensureNoPrec stx; `(ParserDescr.parser $(quote c))
10760     | cs@(_ :: _ :: _) => throwError "ambiguous parser declaration {cs.map (·.1)}"
10761     | [] =>
10762       if Parser.isParserCategory (← getEnv) id then

```

```

10763     processParserCategory stx
10764   else if (← Parser.isParserAlias id) then
10765     ensureNoPrec stx
10766     Parser.ensureConstantParserAlias id
10767     `(ParserDescr.const $(quote id))
10768   else
10769     throwError "unknown parser declaration/category/alias '{id}'"
10770
10771 processUnary (stx : Syntax) := do
10772   let aliasName := (stx[0].getId).eraseMacroScopes
10773   Parser.ensureUnaryParserAlias aliasName
10774   let d ← withNestedParser do process stx[2]
10775   `(ParserDescr.unary $(quote aliasName) $d)
10776
10777 processBinary (stx : Syntax) := do
10778   let aliasName := (stx[0].getId).eraseMacroScopes
10779   Parser.ensureBinaryParserAlias aliasName
10780   let d1 ← withNestedParser do process stx[2]
10781   let d2 ← withNestedParser do process stx[4]
10782   `(ParserDescr.binary $(quote aliasName) $d1 $d2)
10783
10784 processSepBy (stx : Syntax) := do
10785   let p ← withNestedParser $ process stx[1]
10786   let sep := stx[3]
10787   let psep ← if stx[4].isNone then `(ParserDescr.symbol $sep) else process stx[4][1]
10788   let allowTrailingSep := !stx[5].isNone
10789   `(ParserDescr.sepBy $p $sep $psep $(quote allowTrailingSep))
10790
10791 processSepBy1 (stx : Syntax) := do
10792   let p ← withNestedParser do process stx[1]
10793   let sep := stx[3]
10794   let psep ← if stx[4].isNone then `(ParserDescr.symbol $sep) else process stx[4][1]
10795   let allowTrailingSep := !stx[5].isNone
10796   `(ParserDescr.sepBy1 $p $sep $psep $(quote allowTrailingSep))
10797
10798 processAtom (stx : Syntax) := do
10799   match stx[0].isStrLit? with
10800   | some atom =>
10801     /- For syntax categories where initialized with `LeadingIdentBehavior` different from default (e.g., `tactic`), we automatically use
10802     the first symbol as nonReserved. -/
10803     if (← read).behavior != Parser.LeadIdentBehavior.default && (← read).first then
10804       `(ParserDescr.nonReservedSymbol $(quote atom) false)
10805     else
10806       `(ParserDescr.symbol $(quote atom))
10807   | none => throwUnsupportedSyntax
10808
10809 processNonReserved (stx : Syntax) := do

```

```

10810     match stx[1].isStrLit? with
10811     | some atom => `(ParserDescr.nonReservedSymbol $(quote atom) false)
10812     | none      => throwUnsupportedSyntax
10813
10814
10815 end Term
10816
10817 namespace Command
10818 open Lean.Syntax
10819 open Lean.Parser.Term hiding macroArg
10820 open Lean.Parser.Command
10821
10822 private def getCatSuffix (catName : Name) : String :=
10823   match catName with
10824   | Name.str _ s _ => s
10825   | _              => unreachable!
10826
10827 private def declareSyntaxCatQuotParser (catName : Name) : CommandElabM Unit := do
10828   let quotSymbol := "`(" ++ getCatSuffix catName ++ "|"
10829   let name       := catName ++ `quot
10830   -- TODO(Sebastian): this might confuse the pretty printer, but it lets us reuse the elaborator
10831   let kind := ``Lean.Parser.Term.quot
10832   let cmd ← `(
10833     @[termParser] def $(mkIdent name) : Lean.ParserDescr :=
10834       Lean.ParserDescr.node $(quote kind) $(quote Lean.Parser.maxPrec)
10835         (Lean.ParserDescr.binary `andthen (Lean.ParserDescr.symbol $(quote quotSymbol))
10836         (Lean.ParserDescr.binary `andthen (Lean.ParserDescr.cat $(quote catName) 0) (Lean.ParserDescr.symbol ")"))))
10837   elabCommand cmd
10838
10839 @[builtinCommandElab syntaxCat] def elabDeclareSyntaxCat : CommandElab := fun stx => do
10840   let catName := stx[1].getId
10841   let attrName := catName.appendAfter "Parser"
10842   let env ← getEnv
10843   let env ← liftIO $ Parser.registerParserCategory env attrName catName
10844   setEnv env
10845   declareSyntaxCatQuotParser catName
10846
10847 /--
10848   Auxiliary function for creating declaration names from parser descriptions.
10849   Example:
10850   Given
10851   ```
10852   syntax term "+" term : term
10853   syntax "[" sepBy(term, ", ") "]" : term
10854   ```
10855   It generates the names `term_+_` and `term[_`,`
10856 -/

```

```

10857 partial def mkNameFromParserSyntax (catName : Name) (stx : Syntax) : CommandElabM Name :=
10858   mkUnusedBaseName <| Name.mkSimple <| appendCatName <| visit stx ""
10859 where
10860   visit (stx : Syntax) (acc : String) : String :=
10861     match stx.isStrLit? with
10862     | some val => acc ++ (val.trim.map fun c => if c.isWhitespace then '_' else c).capitalize
10863     | none =>
10864       match stx with
10865       | Syntax.node k args =>
10866         if k == `Lean.Parser.Syntax.cat then
10867           acc ++ "_"
10868         else
10869           args.foldl (init := acc) fun acc arg => visit arg acc
10870       | Syntax.ident ..      => acc
10871       | Syntax.atom ..       => acc
10872       | Syntax.missing ..    => acc
10873
10874   appendCatName (str : String) :=
10875     match catName with
10876     | Name.str _ s _ => s ++ str
10877     | _ => str
10878
10879   /- We assume a new syntax can be treated as an atom when it starts and ends with a token.
10880   Here are examples of atom-like syntax.
10881   ```
10882   syntax "(" term ")" : term
10883   syntax "[" (sepBy term ",") "]" : term
10884   syntax "foo" : term
10885   ```
10886   -/
10887 private partial def isAtomLikeSyntax (stx : Syntax) : Bool :=
10888   let kind := stx.getKind
10889   if kind == nullKind then
10890     isAtomLikeSyntax stx[0] && isAtomLikeSyntax stx[stx.getNumArgs - 1]
10891   else if kind == choiceKind then
10892     isAtomLikeSyntax stx[0] -- see toParserDescr
10893   else if kind == `Lean.Parser.Syntax.paren then
10894     isAtomLikeSyntax stx[1]
10895   else
10896     kind == `Lean.Parser.Syntax.atom
10897
10898 @[builtinCommandElab «syntax»] def elabSyntax : CommandElab := fun stx => do
10899   let `($attrKind:attrKind syntax $[: $prec? ]? $[(name := $name?)]? $[(priority := $prio?)]? $[$ps:stx]* : $catStx) ← pure stx
10900   | throwUnsupportedSyntax
10901   let cat := catStx.getId.eraseMacroScopes
10902   unless (Parser.isParserCategory (← getEnv) cat) do
10903     throwErrorAt catStx "unknown category '{cat}'"

```



```

10904 let syntaxParser := mkNullNode ps
10905 -- If the user did not provide an explicit precedence, we assign `maxPrec` to atom-like syntax and `leadPrec` otherwise.
10906 let precDefault := if isAtomLikeSyntax syntaxParser then Parser.maxPrec else Parser.leadPrec
10907 let prec ← match prec? with
10908   | some prec => liftMacroM <| evalPrec prec
10909   | none      => precDefault
10910 let name ← match name? with
10911   | some name => pure name.getId
10912   | none      => mkNameFromParserSyntax cat syntaxParser
10913 let prio ← liftMacroM <| evalOptPrio prio?
10914 let stxNodeKind := (← getCurrNamespace) ++ name
10915 let catParserId := mkIdentFrom stx (cat.appendAfter "Parser")
10916 let (val, lhsPrec?) ← runTermElabM none fun _ => Term.toParserDescr syntaxParser cat
10917 let declName := mkIdentFrom stx name
10918 let d ←
10919   if let some lhsPrec := lhsPrec? then
10920     `(@[$attrKind:attrKind $catParserId:ident $(quote prio):numLit] def $declName : Lean.TrailingParserDescr :=
10921       ParserDescr.trailingNode $(quote stxNodeKind) $(quote prec) $(quote lhsPrec) $val)
10922   else
10923     `(@[$attrKind:attrKind $catParserId:ident $(quote prio):numLit] def $declName : Lean.ParserDescr :=
10924       ParserDescr.node $(quote stxNodeKind) $(quote prec) $val)
10925 trace `Elab fun _ => d
10926 withMacroExpansion stx d <| elabCommand d
10927
10928 /-
10929 def syntaxAbbrev := leading_parser "syntax " >> ident >> " := " >> many1 syntaxParser
10930 -/
10931 @[builtinCommandElab «syntaxAbbrev»] def elabSyntaxAbbrev : CommandElab := fun stx => do
10932   let declName := stx[1]
10933   -- TODO: nonatomic names
10934   let (val, _) ← runTermElabM none $ fun _ => Term.toParserDescr stx[3] Name.anonymous
10935   let stxNodeKind := (← getCurrNamespace) ++ declName.getId
10936   let stx' ← `(def $declName : Lean.ParserDescr := ParserDescr.nodeWithAntiquot $(quote (toString declName.getId)) $(quote stxNodeKind)
10937     withMacroExpansion stx stx' $ elabCommand stx'
10938
10939 private def checkRuleKind (given expected : SyntaxNodeKind) : Bool :=
10940   given == expected || given == expected ++ `antiquot
10941
10942 /-
10943 Remark: `k` is the user provided kind with the current namespace included.
10944 Recall that syntax node kinds contain the current namespace.
10945 -/
10946 def elabMacroRulesAux (k : SyntaxNodeKind) (alts : Array Syntax) : CommandElabM Syntax := do
10947   let alts ← alts.mapM fun alt => match alt with
10948   | `(matchAltExpr | $pats,* => $rhs) => do
10949     let pat := pats.elemsAndSeps[0]
10950     if !pat.isQuot then

```

```

10951     throwUnsupportedSyntax
10952   let quoted := getQuotContent pat
10953   let k' := quoted.getKind
10954   if checkRuleKind k' k then
10955     pure alt
10956   else if k' == choiceKind then
10957     match quoted.getArgs.find? fun quotAlt => checkRuleKind quotAlt.getKind k with
10958     | none => throwErrorAt alt "invalid macro_rules alternative, expected syntax node kind '{k}'"
10959     | some quoted =>
10960       let pat := pat.setArg 1 quoted
10961       let pats := pats.elemsAndSeps.set! 0 pat
10962       `(matchAltExpr | | $pats,* => $rhs)
10963   else
10964     throwErrorAt alt "invalid macro_rules alternative, unexpected syntax node kind '{k}'"
10965 | _ => throwUnsupportedSyntax
10966 `([macro $(Lean.mkIdent k)] def myMacro : Macro :=
10967   fun $alts:matchAlt* | _ => throw Lean.Macro.Exception.unsupportedSyntax)
10968
10969 def inferMacroRulesAltKind : Syntax → CommandElabM SyntaxNodeKind
10970 | `(matchAltExpr | | $pats,* => $rhs) => do
10971   let pat := pats.elemsAndSeps[0]
10972   if !pat.isQuot then
10973     throwUnsupportedSyntax
10974   let quoted := getQuotContent pat
10975   pure quoted.getKind
10976 | _ => throwUnsupportedSyntax
10977
10978 def elabNoKindMacroRulesAux (alts : Array Syntax) : CommandElabM Syntax := do
10979   let mut k ← inferMacroRulesAltKind alts[0]
10980   if k.isStr && k.getString! == "antiquot" then
10981     k := k.getPrefix
10982   if k == choiceKind then
10983     throwErrorAt alts[0]
10984     "invalid macro_rules alternative, multiple interpretations for pattern (solution: specify node kind using `macro_rules [<kind>] ."
10985   else
10986     let altsK ← alts.filterM fun alt => return checkRuleKind (← inferMacroRulesAltKind alt) k
10987     let altsNotK ← alts.filterM fun alt => return !checkRuleKind (← inferMacroRulesAltKind alt) k
10988     let defCmd ← elabMacroRulesAux k altsK
10989     if altsNotK.isEmpty then
10990       pure defCmd
10991     else
10992       `($defCmd:command macro_rules $altsNotK:matchAlt*)
10993
10994 @[builtinCommandElab «macro_rules»] def elabMacroRules : CommandElab :=
10995   adaptExpander fun stx => match stx with
10996   | `(macro_rules $alts:matchAlt*) => elabNoKindMacroRulesAux alts
10997   | `(macro_rules (kind := $kind) | $x:ident => $rhs) => `([macro $kind] def myMacro : Macro := fun $x:ident => $rhs)

```

```

10998 | `(macro_rules (kind := $kind) $alts:matchAlt*)    => do elabMacroRulesAux ((← getCurrNamespace) ++ kind.getId) alts
10999 | _                                                  => throwUnsupportedSyntax
11000
11001 @[builtinMacro Lean.Parser.Command.mixfix] def expandMixfix : Macro := fun stx =>
11002   withAttrKindGlobal stx fun stx => do
11003     match stx with
11004     | `(infixl $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
11005       let prec1 := quote <| (← evalOptPrec prec) + 1
11006       `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs$[:$prec]? $op:strLit rhs:$prec1 => $f lhs rhs)
11007     | `(infix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
11008       let prec1 := quote <| (← evalOptPrec prec) + 1
11009       `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs:$prec1 $op:strLit rhs:$prec1 => $f lhs rhs)
11010     | `(infixr $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
11011       let prec1 := quote <| (← evalOptPrec prec) + 1
11012       `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? lhs:$prec1 $op:strLit rhs $[: $prec]? => $f lhs rhs)
11013     | `(prefix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
11014       `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op:strLit arg $[: $prec]? => $f arg)
11015     | `(postfix $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? $op => $f) =>
11016       `(notation $[: $prec]? $[(name := $name)]? $[(priority := $prio)]? arg$[:$prec]? $op:strLit => $f arg)
11017     | _ => Macro.throwUnsupported
11018 where
11019   -- set "global" `attrKind`, apply `f`, and restore `attrKind` to result
11020   withAttrKindGlobal stx f := do
11021     let attrKind := stx[0]
11022     let stx := stx.setArg 0 mkAttrKindGlobal
11023     let stx ← f stx
11024     return stx.setArg 0 attrKind
11025
11026 /- Wrap all occurrences of the given `ident` nodes in antiquotations -/
11027 private partial def antiquote (vars : Array Syntax) : Syntax → Syntax
11028 | stx => match stx with
11029 | `($id:ident) =>
11030   if (vars.findIdx? (fun var => var.getId == id.getId)).isSome then
11031     mkAntiquotNode id
11032   else
11033     stx
11034 | _ => match stx with
11035 | Syntax.node k args => Syntax.node k (args.map (antiquote vars))
11036 | stx => stx
11037
11038 /- Convert `notation` command lhs item into a `syntax` command item -/
11039 def expandNotationItemIntoSyntaxItem (stx : Syntax) : CommandElabM Syntax :=
11040   let k := stx.getKind
11041   if k == `Lean.Parser.Command.identPrec then
11042     pure $ Syntax.node `Lean.Parser.Syntax.cat #[mkIdentFrom stx `term, stx[1]]
11043   else if k == strLitKind then
11044     pure $ Syntax.node `Lean.Parser.Syntax.atom #[stx]

```

```

11045 else
11046   throwUnsupportedSyntax
11047
11048 def strLitToPattern (stx: Syntax) : MacroM Syntax :=
11049   match stx.isStrLit? with
11050   | some str => pure $ mkAtomFrom stx str
11051   | none     => Macro.throwUnsupported
11052
11053 /- Convert `notation` command lhs item into a pattern element -/
11054 def expandNotationItemIntoPattern (stx : Syntax) : CommandElabM Syntax :=
11055   let k := stx.getKind
11056   if k == `Lean.Parser.Command.identPrec then
11057     mkAntiquotNode stx[0]
11058   else if k == strLitKind then
11059     liftMacroM <| strLitToPattern stx
11060   else
11061     throwUnsupportedSyntax
11062
11063 /-- Try to derive a `SimpleDelab` from a notation.
11064 The notation must be of the form `notation ... => c var_1 ... var_n`
11065 where `c` is a declaration in the current scope and the `var_i` are a permutation of the LHS vars. -/
11066 def mkSimpleDelab (attrKind : Syntax) (vars : Array Syntax) (pat qrhs : Syntax) : OptionT CommandElabM Syntax := do
11067   match qrhs with
11068   | `($c:ident $args*) =>
11069     let [(c, [])] ← resolveGlobalName c.getId | failure
11070     guard <| args.all (Syntax.isIdent . getAntiquotTerm)
11071     guard <| args.allDiff
11072     -- replace head constant with (unused) antiquotation so we're not dependent on the exact pretty printing of the head
11073     let qrhs ← `($ (mkAntiquotNode (← `(_))) $args*)
11074     `(@[$attrKind:attrKind appUnexpander $(mkIdent c):ident] def unexpand : Lean.PrettyPrinter.Unexpander := fun
11075       | `($qrhs) => `($pat)
11076       | _        => throw ())
11077   | `($c:ident) =>
11078     let [(c, [])] ← resolveGlobalName c.getId | failure
11079     `(@[$attrKind:attrKind appUnexpander $(mkIdent c):ident] def unexpand : Lean.PrettyPrinter.Unexpander := fun _ => `($pat))
11080   | _ => failure
11081
11082 private def expandNotationAux (ref : Syntax)
11083   (currNamespace : Name) (attrKind : Syntax) (prec? : Option Syntax) (name? : Option Syntax) (prio? : Option Syntax) (items : Array S
11084   let prio ← liftMacroM <| evalOptPrio prio?
11085   -- build parser
11086   let syntaxParts ← items.mapM expandNotationItemIntoSyntaxItem
11087   let cat := mkIdentFrom ref `term
11088   let name ←
11089     match name? with
11090     | some name => pure name.getId
11091     | none      => mkNameFromParserSyntax `term (mkNullNode syntaxParts)

```

```

11092 -- build macro rules
11093 let vars := items.filter fun item => item.getKind == `Lean.Parser.Command.identPrec
11094 let vars := vars.map fun var => var[0]
11095 let qrhs := antiquote vars rhs
11096 let patArgs ← items.mapM expandNotationItemIntoPattern
11097 /- The command `syntax [<kind>] ...` adds the current namespace to the syntax node kind.
11098 So, we must include current namespace when we create a pattern for the following `macro_rules` commands. -/
11099 let fullName := currNamespace ++ name
11100 let pat := Syntax.node fullName patArgs
11101 let stxDecl ← `($attrKind:attrKind syntax $[: $prec?]? (name := $(mkIdent name)) (priority := $(quote prio):numLit) $[$syntaxParts]*
11102 let macroDecl ← `(macro_rules | `($pat) => `($qrhs))
11103 match (← mkSimpleDelab attrKind vars pat qrhs |>.run) with
11104 | some delabDecl => mkNullNode #[stxDecl, macroDecl, delabDecl]
11105 | none           => mkNullNode #[stxDecl, macroDecl]
11106
11107 @[builtinCommandElab «notation»] def expandNotation : CommandElab :=
11108   adaptExpander fun stx => do
11109     let currNamespace ← getCurrNamespace
11110     match stx with
11111     | `($attrKind:attrKind notation $[: $prec? ]? $[(name := $name?)]? $[(priority := $prio?)]? $items* => $rhs) =>
11112       -- trigger scoped checks early and only once
11113       let _ ← toAttributeKind attrKind
11114       expandNotationAux stx currNamespace attrKind prec? name? prio? items rhs
11115     | _ => throwUnsupportedSyntax
11116
11117 /- Convert `macro` argument into a `syntax` command item -/
11118 def expandMacroArgIntoSyntaxItem : Macro
11119 | `(macroArg|$id:ident:$stx) => stx
11120 -- can't match against `$s:strLit%$id` because the latter part would be interpreted as an antiquotation on the token
11121 -- `strLit`.
11122 | `(macroArg|$s:macroArgSymbol) => `(stx|$(s[0]):strLit)
11123 | _ => Macro.throwUnsupported
11124
11125 /- Convert `macro` arg into a pattern element -/
11126 def expandMacroArgIntoPattern (stx : Syntax) : MacroM Syntax := do
11127   match (← expandMacros stx) with
11128   | `(macroArg|$id:ident:optional($stx)) =>
11129     mkSplicePat `optional id "?"
11130   | `(macroArg|$id:ident:many($stx)) =>
11131     mkSplicePat `many id "*"
11132   | `(macroArg|$id:ident:many1($stx)) =>
11133     mkSplicePat `many id "*"
11134   | `(macroArg|$id:ident:sepBy($stx, $sep:strLit $[, $stxsep]? $[, allowTrailingSep?]) =>
11135     mkSplicePat `sepBy id ((isStrLit? sep).get! ++ "**")
11136   | `(macroArg|$id:ident:sepBy1($stx, $sep:strLit $[, $stxsep]? $[, allowTrailingSep?]) =>
11137     mkSplicePat `sepBy id ((isStrLit? sep).get! ++ "**")
11138   | `(macroArg|$id:ident:$stx) => mkAntiquotNode id

```

```

11139 | `(macroArg|$:strLit) => strLitToPattern s
11140 -- `tk"%id` ~> `tk"%$id`
11141 | `(macroArg|$:macroArgSymbol) => mkNode `token_antiquot #[← strLitToPattern s[0], mkAtom "%", mkAtom "$", s[1][1]]
11142 | _ => Macro.throwUnsupported
11143 where mkSplicePat kind id suffix :=
11144   mkNullNode #[mkAntiquotSuffixSpliceNode kind (mkAntiquotNode id) suffix]
11145
11146
11147 /- «macro» := leading_parser suppressInsideQuot (Term.attrKind >> "macro " >> optPrecedence >> optNamedName >> optNamedPrio >> macroHead)
11148 def expandMacro (currNamespace : Name) (stx : Syntax) : CommandElabM Syntax := do
11149   let attrKind := stx[0]
11150   let prec := stx[2].getOptional?
11151   let name? ← liftMacroM <| expandOptNamedName stx[3]
11152   let prio ← liftMacroM <| expandOptNamedPrio stx[4]
11153   let head := stx[5]
11154   let args := stx[6].getArgs
11155   let cat := stx[8]
11156   -- build parser
11157   let stxPart ← liftMacroM <| expandMacroArgIntoSyntaxItem head
11158   let stxParts ← liftMacroM <| args.mapM expandMacroArgIntoSyntaxItem
11159   let stxParts := #[stxPart] ++ stxParts
11160   -- name
11161   let name ← match name? with
11162     | some name => pure name
11163     | none => mkNameFromParserSyntax cat.getId (mkNullNode stxParts)
11164   -- build macro rules
11165   let patHead ← liftMacroM <| expandMacroArgIntoPattern head
11166   let patArgs ← liftMacroM <| args.mapM expandMacroArgIntoPattern
11167   /- The command `syntax [<kind>] ...` adds the current namespace to the syntax node kind.
11168   So, we must include current namespace when we create a pattern for the following `macro_rules` commands. -/
11169   let pat := Syntax.node (currNamespace ++ name) (#[patHead] ++ patArgs)
11170   if stx.getArgs.size == 11 then
11171     -- `stx` is of the form `macro $head $args* : $cat => term`
11172     let rhs := stx[10]
11173     let stxCmd ← `(Parser.Command.syntax| $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quo
11174     let macroRulesCmd ← `(macro_rules | `($pat) => $rhs)
11175     return mkNullNode #[stxCmd, macroRulesCmd]
11176   else
11177     -- `stx` is of the form `macro $head $args* : $cat => `( $body )`
11178     let rhsBody := stx[11]
11179     let stxCmd ← `(Parser.Command.syntax| $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quo
11180     let macroRulesCmd ← `(macro_rules | `($pat) => `($rhsBody))
11181     return mkNullNode #[stxCmd, macroRulesCmd]
11182
11183 @[builtinCommandElab «macro»] def elabMacro : CommandElab :=
11184   adaptExpander fun stx => do
11185     expandMacro (← getCurrNamespace) stx

```

```

11186
11187 builtin_initialize
11188   registerTraceClass `Elab.syntax
11189
11190 @[inline] def withExpectedType (expectedType? : Option Expr) (x : Expr → TermElabM Expr) : TermElabM Expr := do
11191   Term.tryPostponeIfNoneOrMVar expectedType?
11192   let some expectedType ← pure expectedType?
11193   | throwError "expected type must be known"
11194   x expectedType
11195
11196 /-
11197 def elabTail := try (" : " >> ident) >> darrow >> termParser
11198 def «elab» := leading_parser suppressInsideQuot (Term.attrKind >> "elab " >> optPrecedence >> optNamedName >> optNamedPrio >> elabHead :
11199 -/
11200 def expandElab (currNamespace : Name) (stx : Syntax) : CommandElabM Syntax := do
11201   let ref := stx
11202   let attrKind := stx[0]
11203   let prec     := stx[2].getOptional?
11204   let name?    ← liftMacroM <| expandOptNamedName stx[3]
11205   let prio     ← liftMacroM <| expandOptNamedPrio stx[4]
11206   let head     := stx[5]
11207   let args     := stx[6].getArgs
11208   let cat      := stx[8]
11209   let expectedTypeSpec := stx[9]
11210   let rhs      := stx[11]
11211   let catName  := cat.getId
11212   -- build parser
11213   let stxPart  ← liftMacroM <| expandMacroArgIntoSyntaxItem head
11214   let stxParts ← liftMacroM <| args.mapM expandMacroArgIntoSyntaxItem
11215   let stxParts := #[stxPart] ++ stxParts
11216   -- name
11217   let name ← match name? with
11218   | some name => pure name
11219   | none => mkNameFromParserSyntax cat.getId (mkNullNode stxParts)
11220   -- build pattern for `match_syntax
11221   let patHead ← liftMacroM <| expandMacroArgIntoPattern head
11222   let patArgs ← liftMacroM <| args.mapM expandMacroArgIntoPattern
11223   let pat := Syntax.node (currNamespace ++ name) (#[patHead] ++ patArgs)
11224   let stxCmd ← `(Parser.Command.syntax|
11225     $attrKind:attrKind syntax $(prec)? (name := $(mkIdentFrom stx name):ident) (priority := $(quote prio):numLit) $[stxParts]* : $cat)
11226   let elabCmd ←
11227     if expectedTypeSpec.hasArgs then
11228       if catName == `term then
11229         let expId := expectedTypeSpec[1]
11230         `([termElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Term.TermElab :=
11231           fun stx expectedType? => match stx with
11232           | `($pat) => Lean.Elab.Command.withExpectedType expectedType? fun $expId => $rhs

```

```

11233         | _ => throwUnsupportedSyntax)
11234     else
11235         throwErrorAt expectedTypeSpec "syntax category '{catName}' does not support expected type specification"
11236     else if catName == `term then
11237         `([termElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Term.TermElab :=
11238             fun stx _ => match stx with
11239             | `($pat) => $rhs
11240             | _ => throwUnsupportedSyntax)
11241     else if catName == `command then
11242         `([commandElab $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Command.CommandElab :=
11243             fun
11244             | `($pat) => $rhs
11245             | _ => throwUnsupportedSyntax)
11246     else if catName == `tactic then
11247         `([tactic $(mkIdentFrom stx name):ident] def elabFn : Lean.Elab.Tactic.Tactic :=
11248             fun
11249             | `(tactic|$pat) => $rhs
11250             | _ => throwUnsupportedSyntax)
11251     else
11252         -- We considered making the command extensible and support new user-defined categories. We think it is unnecessary.
11253         -- If users want this feature, they add their own `elab` macro that uses this one as a fallback.
11254         throwError "unsupported syntax category '{catName}'"
11255     return mkNullNode #[stxCmd, elabCmd]
11256
11257 @[builtinCommandElab «elab»] def elabElab : CommandElab :=
11258     adaptExpander fun stx => do
11259         expandElab (← getCurrNamespace) stx
11260
11261 end Lean.Elab.Command
11262 ::::::::::::::
11263 Elab/SyntheticMVars.lean
11264 ::::::::::::::
11265 /-
11266 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
11267 Released under Apache 2.0 license as described in the file LICENSE.
11268 Authors: Leonardo de Moura, Sebastian Ullrich
11269 -/
11270 import Lean.Util.ForEachExpr
11271 import Lean.Elab.Term
11272 import Lean.Elab.Tactic.Basic
11273
11274 namespace Lean.Elab.Term
11275 open Tactic (TacticM evalTactic getUnsolvedGoals)
11276 open Meta
11277
11278 /- Auxiliary function used to implement `synthesizeSyntheticMVars`. -/
11279 private def resumeElabTerm (stx : Syntax) (expectedType? : Option Expr) (errToSorry := true) : TermElabM Expr :=

```



```

11280 -- Remark: if `ctx.errToSorry` is already false, then we don't enable it. Recall tactics disable `errToSorry`
11281 withReader (fun ctx => { ctx with errToSorry := ctx.errToSorry && errToSorry }) do
11282   elabTerm stx expectedType? false
11283
11284 /--
11285   Try to elaborate `stx` that was postponed by an elaboration method using `Expectation.postpone`.
11286   It returns `true` if it succeeded, and `false` otherwise.
11287   It is used to implement `synthesizeSyntheticMVars`. -/
11288 private def resumePostponed (savedContext : SavedContext) (stx : Syntax) (mvarId : MVarId) (postponeOnError : Bool) : TermElabM Bool :=
11289   withRef stx <| withMVarContext mvarId do
11290     let s ← get
11291     try
11292       withSavedContext savedContext do
11293         let mvarDecl ← getMVarDecl mvarId
11294         let expectedType ← instantiateMVars mvarDecl.type
11295         withInfoHole mvarId do
11296           let result ← resumeElabTerm stx expectedType (!postponeOnError)
11297           /- We must ensure `result` has the expected type because it is the one expected by the method that postponed stx.
11298              That is, the method does not have an opportunity to check whether `result` has the expected type or not. -/
11299           let result ← withRef stx <| ensureHasType expectedType result
11300           /- We must perform `occursCheck` here since `result` may contain `mvarId` when it has synthetic `sorry`s. -/
11301           if (← occursCheck mvarId result) then
11302             assignExprMVar mvarId result
11303             return true
11304           else
11305             return false
11306       catch
11307         | ex@(Exception.internal id _) =>
11308           if id == postponeExceptionId then
11309             set s
11310             return false
11311           else
11312             throw ex
11313         | ex@(Exception.error _ _) =>
11314           if postponeOnError then
11315             set s
11316             return false
11317           else
11318             logException ex
11319             return true
11320
11321 /--
11322   Similar to `synthesizeInstMVarCore`, but makes sure that `instMVar` local context and instances
11323   are used. It also logs any error message produced. -/
11324 private def synthesizePendingInstMVar (instMVar : MVarId) : TermElabM Bool :=
11325   withMVarContext instMVar do
11326     try

```

```

11327     synthesizeInstMVarCore instMVar
11328   catch
11329   | ex@(Exception.error _ _) => logException ex; return true
11330   | _                       => unreachable!
11331
11332 /--
11333   Similar to `synthesizePendingInstMVar`, but generates type mismatch error message.
11334   Remark: `eNew` is of the form `@coe ... mvar`, where `mvar` is the metavariable for the `CoeT ...` instance.
11335   If `mvar` can be synthesized, then assign `auxMVarId := (expandCoe eNew)`.
11336 -/
11337 private def synthesizePendingCoeInstMVar
11338   (auxMVarId : MVarId) (errorMsgHeader? : Option String) (eNew : Expr) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr) :
11339   let instMVarId := eNew.appArg!.mvarId!
11340   withMVarContext instMVarId do
11341     if (← isDefEq expectedType eType) then
11342       /- This case may seem counterintuitive since we created the coercion
11343          because the `isDefEq expectedType eType` test failed before.
11344          However, it may succeed here because we have more information, for example, metavariables
11345          occurring at `expectedType` and `eType` may have been assigned. -/
11346       if (← occursCheck auxMVarId e) then
11347         assignExprMVar auxMVarId e
11348         return true
11349       else
11350         return false
11351     try
11352       if (← synthesizeCoeInstMVarCore instMVarId) then
11353         let eNew ← expandCoe eNew
11354         if (← occursCheck auxMVarId eNew) then
11355           assignExprMVar auxMVarId eNew
11356           return true
11357         return false
11358     catch
11359     | Exception.error _ msg => throwTypeMismatchError errorMsgHeader? expectedType eType e f? msg
11360     | _                     => unreachable!
11361
11362 /--
11363   Try to synthesize a value for `mvarId` using the given default instance.
11364   Return `some (val, mvarDecls)` if successful, where `val` is the value assigned to `mvarId`, and `mvarDecls` is a list of new type cl.
11365 -/
11366 private def tryToSynthesizeUsingDefaultInstance (mvarId : MVarId) (defaultInstance : Name) : TermElabM (Option (Expr × List SyntheticMVar)) :
11367   commitWhenSome? do
11368     let candidate ← mkConstWithFreshMVarLevels defaultInstance
11369     let (mvars, bis, _) ← forallMetaTelescopeReducing (← inferType candidate)
11370     let candidate := mkAppN candidate mvars
11371     trace[Elab.resume] "trying default instance for {mkMVar mvarId} := {candidate}"
11372     if (← isDefEqGuarded (mkMVar mvarId) candidate) then
11373       -- Succeeded. Collect new TC problems

```

```

11374     let mut result := []
11375     for i in [:bis.size] do
11376       if bis[i] == BinderInfo.instImplicit then
11377         result := { mvarId := mvars[i].mvarId!, stx := (← getRef), kind := SyntheticMVarKind.typeClass } :: result
11378       trace[Elab.resume] "worked"
11379       return some (candidate, result)
11380     else
11381       return none
11382
11383 private def tryToSynthesizeUsingDefaultInstances (mvarId : MVarId) (prio : Nat) : TermElabM (Option (Expr × List SyntheticMVarDecl)) :=
11384   withMVarContext mvarId do
11385     let mvarType := (← Meta.getMVarDecl mvarId).type
11386     match (← isClass? mvarType) with
11387     | none => return none
11388     | some className =>
11389       match (← getDefaultInstances className) with
11390       | [] => return none
11391       | defaultInstances =>
11392         for (defaultInstance, instPrio) in defaultInstances do
11393           if instPrio == prio then
11394             match (← tryToSynthesizeUsingDefaultInstance mvarId defaultInstance) with
11395             | some result => return some result
11396             | none => continue
11397         return none
11398
11399 /- Used to implement `synthesizeUsingDefault`. This method only consider default instances with the given priority. -/
11400 private def synthesizeUsingDefaultPrio (prio : Nat) : TermElabM Bool := do
11401   let rec visit (syntheticMVars : List SyntheticMVarDecl) (syntheticMVarsNew : List SyntheticMVarDecl) : TermElabM Bool := do
11402     match syntheticMVars with
11403     | [] => return false
11404     | mvarDecl :: mvarDecls =>
11405       match mvarDecl.kind with
11406       | SyntheticMVarKind.typeClass =>
11407         match (← withRef mvarDecl.stx <| tryToSynthesizeUsingDefaultInstances mvarDecl.mvarId prio) with
11408         | none => visit mvarDecls (mvarDecl :: syntheticMVarsNew)
11409         | some (val, newMVarDecls) =>
11410           for newMVarDecl in newMVarDecls do
11411             -- Register that `newMVarDecl.mvarId`s are implicit arguments of the value assigned to `mvarDecl.mvarId`
11412             registerMVarErrorImplicitArgInfo newMVarDecl.mvarId (← getRef) val
11413           let syntheticMVarsNew := newMVarDecls ++ syntheticMVarsNew
11414           let syntheticMVarsNew := mvarDecls.reverse ++ syntheticMVarsNew
11415           modify fun s => { s with syntheticMVars := syntheticMVarsNew }
11416           return true
11417       | _ => visit mvarDecls (mvarDecl :: syntheticMVarsNew)
11418 /- Recall that s.syntheticMVars is essentially a stack. The first metavariable was the last one created.
11419 We want to apply the default instance in reverse creation order. Otherwise,
11420 `toString 0` will produce a `OfNat String _` cannot be synthesized error. -/

```

```

11421 visit (← get).syntheticMVars.reverse []
11422
11423 /--
11424   Apply default value to any pending synthetic metavariable of kind `SyntheticMVarKind.withDefault`
11425   Return true if something was synthesized. -/
11426 private def synthesizeUsingDefault : TermElabM Bool := do
11427   let prioSet ← getDefaultInstancesPriorities
11428   /- Recall that `prioSet` is stored in descending order -/
11429   for prio in prioSet do
11430     if (← synthesizeUsingDefaultPrio prio) then
11431       return true
11432   return false
11433
11434 /-- Report an error for each synthetic metavariable that could not be resolved. -/
11435 private def reportStuckSyntheticMVars : TermElabM Unit := do
11436   let syntheticMVars ← modifyGet fun s => (s.syntheticMVars, { s with syntheticMVars := [] })
11437   for mvarSyntheticDecl in syntheticMVars do
11438     withRef mvarSyntheticDecl.stx do
11439       match mvarSyntheticDecl.kind with
11440       | SyntheticMVarKind.typeClass =>
11441         withMVarContext mvarSyntheticDecl.mvarId do
11442           let mvarDecl ← getMVarDecl mvarSyntheticDecl.mvarId
11443           unless (← get).messages.hasErrors do
11444             throwError "typeclass instance problem is stuck, it is often due to metavariables{indentExpr mvarDecl.type}"
11445       | SyntheticMVarKind.coe header eNew expectedType eType e f? =>
11446         let mvarId := eNew.appArg!.mvarId!
11447         withMVarContext mvarId do
11448           let mvarDecl ← getMVarDecl mvarId
11449           throwTypeMismatchError header expectedType eType e f? (some ("failed to create type class instance for " ++ indentExpr mvarDecl
11450           | _ => unreachable! -- TODO handle other cases.
11451
11452 private def getSomeSyntheticMVarsRef : TermElabM Syntax := do
11453   let s ← get
11454   match s.syntheticMVars.find? fun (mvarDecl : SyntheticMVarDecl) => !mvarDecl.stx.getPos?.isNone with
11455   | some mvarDecl => return mvarDecl.stx
11456   | none          => return Syntax.missing
11457
11458 mutual
11459
11460 partial def liftTacticElabM {α} (mvarId : MVarId) (x : TacticM α) : TermElabM α :=
11461   withMVarContext mvarId do
11462     let savedSyntheticMVars := (← get).syntheticMVars
11463     modify fun s => { s with syntheticMVars := [] }
11464     try
11465       let a ← x.run' { main := mvarId } { goals := [mvarId] }
11466       synthesizeSyntheticMVars (mayPostpone := false)
11467       pure a

```

```

11468     finally
11469         modify fun s => { s with syntheticMVars := savedSyntheticMVars }
11470
11471 partial def runTactic (mvarId : MVarId) (tacticCode : Syntax) : TermElabM Unit := do
11472     /- Recall, `tacticCode` is the whole `by ...` expression.
11473     We store the `by` because in the future we want to save the initial state information at the `by` position. -/
11474     let code := tacticCode[1]
11475     modifyThe Meta.State fun s => { s with mctx := s.mctx.instantiateMVarDeclMVars mvarId }
11476     let remainingGoals ← withInfoHole mvarId do liftTacticElabM mvarId do evalTactic code; getUnsolvedGoals
11477     unless remainingGoals.isEmpty do reportUnsolvedGoals remainingGoals
11478
11479 /-- Try to synthesize the given pending synthetic metavariable. -/
11480 private partial def synthesizeSyntheticMVar (mvarSyntheticDecl : SyntheticMVarDecl) (postponeOnError : Bool) (runTactics : Bool) : Te
11481     withRef mvarSyntheticDecl.stx do
11482         match mvarSyntheticDecl.kind with
11483         | SyntheticMVarKind.typeClass => synthesizePendingInstMVar mvarSyntheticDecl.mvarId
11484         | SyntheticMVarKind.coe header? eNew expectedType eType e f? => synthesizePendingCoeInstMVar mvarSyntheticDecl.mvarId header? eNew
11485         -- NOTE: actual processing at `synthesizeSyntheticMVarsAux`
11486         | SyntheticMVarKind.postponed savedContext => resumePostponed savedContext mvarSyntheticDecl.stx mvarSyntheticDecl.mvarId postponeO
11487         | SyntheticMVarKind.tactic tacticCode savedContext =>
11488             withSavedContext savedContext do
11489                 if runTactics then
11490                     runTactic mvarSyntheticDecl.mvarId tacticCode
11491                     return true
11492                 else
11493                     return false
11494 /--
11495     Try to synthesize the current list of pending synthetic metavariables.
11496     Return `true` if at least one of them was synthesized. -/
11497 private partial def synthesizeSyntheticMVarsStep (postponeOnError : Bool) (runTactics : Bool) : TermElabM Bool := do
11498     let ctx ← read
11499     traceAtCmdPos `Elab.resuming fun _ =>
11500         m!"resuming synthetic metavariables, mayPostpone: {ctx.mayPostpone}, postponeOnError: {postponeOnError}"
11501     let syntheticMVars := (← get).syntheticMVars
11502     let numSyntheticMVars := syntheticMVars.length
11503     -- We reset `syntheticMVars` because new synthetic metavariables may be created by `synthesizeSyntheticMVar`.
11504     modify fun s => { s with syntheticMVars := [] }
11505     -- Recall that `syntheticMVars` is a list where head is the most recent pending synthetic metavariable.
11506     -- We use `filterRevM` instead of `filterM` to make sure we process the synthetic metavariables using the order they were created.
11507     -- It would not be incorrect to use `filterM`.
11508     let remainingSyntheticMVars ← syntheticMVars.filterRevM fun mvarDecl => do
11509         -- We use `traceM` because we want to make sure the metavar local context is used to trace the message
11510         traceM `Elab.postpone (withMVarContext mvarDecl.mvarId do addMessageContext m!"resuming {mkMVar mvarDecl.mvarId}")
11511         let succeeded ← synthesizeSyntheticMVar mvarDecl postponeOnError runTactics
11512         trace[Elab.postpone] if succeeded then fmt "succeeded" else fmt "not ready yet"
11513         pure !succeeded
11514     -- Merge new synthetic metavariables with `remainingSyntheticMVars`, i.e., metavariables that still couldn't be synthesized

```

```

11515 modify fun s => { s with syntheticMVars := s.syntheticMVars ++ remainingSyntheticMVars }
11516 return numSyntheticMVars != remainingSyntheticMVars.length
11517
11518 /--
11519   Try to process pending synthetic metavariables. If `mayPostpone == false`,
11520   then `syntheticMVars` is `[]` after executing this method.
11521
11522   It keeps executing `synthesizeSyntheticMVarsStep` while progress is being made.
11523   If `mayPostpone == false`, then it applies default instances to `SyntheticMVarKind.typeClass` (if available)
11524   metavariables that are still unresolved, and then tries to resolve metavariables
11525   with `mayPostpone == false`. That is, we force them to produce error messages and/or commit to
11526   a "best option". If, after that, we still haven't made progress, we report "stuck" errors. -/
11527 partial def synthesizeSyntheticMVars (mayPostpone := true) : TermElabM Unit :=
11528   let rec loop (u : Unit) : TermElabM Unit := do
11529     withRef (← getSomeSyntheticMVarsRef) <| withIncRecDepth do
11530       unless (← get).syntheticMVars.isEmpty do
11531         if ← synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := false) then
11532           loop ()
11533         else if !mayPostpone then
11534           /- Resume pending metavariables with "elaboration postponement" disabled.
11535              We postpone elaboration errors in this step by setting `postponeOnError := true`.
11536              Example:
11537              ```
11538              #check let x := {1, 2}; Prod.fst x
11539              ```
11540              The term `{1, 2}` can't be elaborated because the expected type is not known.
11541              The `x` at `Prod.fst x` is not elaborated because the type of `x` is not known.
11542              When we execute the following step with "elaboration postponement" disabled,
11543              the elaborator fails at `{1, 2}` and postpones it, and succeeds at `x` and learns
11544              that its type must be of the form `Prod ?α ?β`.
11545
11546              Recall that we postponed `x` at `Prod.fst x` because its type it is not known.
11547              We the type of `x` may learn later its type and it may contain implicit and/or auto arguments.
11548              By disabling postponement, we are essentially giving up the opportunity of learning `x`'s type
11549              and assume it does not have implicit and/or auto arguments. -/
11550           if ← withoutPostponing <| synthesizeSyntheticMVarsStep (postponeOnError := true) (runTactics := false) then
11551             loop ()
11552           else if ← synthesizeUsingDefault then
11553             loop ()
11554           else if ← withoutPostponing <| synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := false) then
11555             loop ()
11556           else if ← synthesizeSyntheticMVarsStep (postponeOnError := false) (runTactics := true) then
11557             loop ()
11558           else
11559             reportStuckSyntheticMVars
11560       loop ()
11561 end

```

```

11562
11563 def synthesizeSyntheticMVarsNoPostponing : TermElabM Unit :=
11564   synthesizeSyntheticMVars (mayPostpone := false)
11565
11566 /- Keep invoking `synthesizeUsingDefault` until it returns false. -/
11567 private partial def synthesizeUsingDefaultLoop : TermElabM Unit := do
11568   if (← synthesizeUsingDefault) then
11569     synthesizeSyntheticMVars (mayPostpone := true)
11570     synthesizeUsingDefaultLoop
11571
11572 def synthesizeSyntheticMVarsUsingDefault : TermElabM Unit := do
11573   synthesizeSyntheticMVars (mayPostpone := true)
11574   synthesizeUsingDefaultLoop
11575
11576 private partial def withSynthesizeImp {α} (k : TermElabM α) (mayPostpone : Bool) : TermElabM α := do
11577   let syntheticMVarsSaved := (← get).syntheticMVars
11578   modify fun s => { s with syntheticMVars := [] }
11579   try
11580     let a ← k
11581     synthesizeSyntheticMVars mayPostpone
11582     if mayPostpone then
11583       synthesizeUsingDefaultLoop
11584     return a
11585   finally
11586     modify fun s => { s with syntheticMVars := s.syntheticMVars ++ syntheticMVarsSaved }
11587
11588 /--
11589   Execute `k`, and synthesize pending synthetic metavariables created while executing `k` are solved.
11590   If `mayPostpone == false`, then all of them must be synthesized.
11591   Remark: even if `mayPostpone == true`, the method still uses `synthesizeUsingDefault` -/
11592 @[inline] def withSynthesize [MonadFunctorT TermElabM m] [Monad m] (k : m α) (mayPostpone := false) : m α :=
11593   monadMap (m := TermElabM) (withSynthesizeImp . mayPostpone) k
11594
11595 /-- Elaborate `stx`, and make sure all pending synthetic metavariables created while elaborating `stx` are solved. -/
11596 def elabTermAndSynthesize (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr :=
11597   withRef stx do
11598     instantiateMVars (← withSynthesize <| elabTerm stx expectedType?)
11599
11600 end Lean.Elab.Term
11601 ::::::::::::::
11602 Elab/Tactic.lean
11603 ::::::::::::::
11604 /-
11605 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
11606 Released under Apache 2.0 license as described in the file LICENSE.
11607 Authors: Leonardo de Moura, Sebastian Ullrich
11608 -/

```

```

11609 import Lean.Elab.Term
11610 import Lean.Elab.Tactic.Basic
11611 import Lean.Elab.Tactic.ElabTerm
11612 import Lean.Elab.Tactic.Induction
11613 import Lean.Elab.Tactic.Generalize
11614 import Lean.Elab.Tactic.Injection
11615 import Lean.Elab.Tactic.Match
11616 import Lean.Elab.Tactic.Rewrite
11617 import Lean.Elab.Tactic.Location
11618 import Lean.Elab.Tactic.Simp
11619 ::::::::::::::
11620 Elab/Term.lean
11621 ::::::::::::::
11622 /-
11623 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
11624 Released under Apache 2.0 license as described in the file LICENSE.
11625 Authors: Leonardo de Moura
11626 -/
11627 import Lean.ResolveName
11628 import Lean.Util.Sorry
11629 import Lean.Util.ReplaceExpr
11630 import Lean.Structure
11631 import Lean.Meta.ExprDefEq
11632 import Lean.Meta.AppBuilder
11633 import Lean.Meta.SynthInstance
11634 import Lean.Meta.CollectMVars
11635 import Lean.Meta.Coe
11636 import Lean.Meta.Tactic.Util
11637 import Lean.Hygiene
11638 import Lean.Util.RecDepth
11639 import Lean.Elab.Log
11640 import Lean.Elab.Level
11641 import Lean.Elab.Attributes
11642 import Lean.Elab.AutoBound
11643 import Lean.Elab.InfoTree
11644 import Lean.Elab.Open
11645 import Lean.Elab.SetOption
11646
11647 namespace Lean.Elab.Term
11648 /-
11649   Set isDefEq configuration for the elaborator.
11650   Note that we enable all approximations but `quasiPatternApprox`
11651
11652   In Lean3 and Lean 4, we used to use the quasi-pattern approximation during elaboration.
11653   The example:
11654   ```
11655   def ex : StateT δ (StateT σ Id) σ :=

```



```

11656 monadLift (get : StateT  $\sigma$  Id  $\sigma$ )
11657 ``
11658 demonstrates why it produces counterintuitive behavior.
11659 We have the `Monad-lift` application:
11660 ``
11661 @monadLift ?m ?n ?c ? $\alpha$  (get : StateT  $\sigma$  id  $\sigma$ ) : ?n ? $\alpha$ 
11662 ``
11663 It produces the following unification problem when we process the expected type:
11664 ``
11665 ?n ? $\alpha$  =?= StateT  $\delta$  (StateT  $\sigma$  id)  $\sigma$ 
11666 ==> (approximate using first-order unification)
11667 ?n := StateT  $\delta$  (StateT  $\sigma$  id)
11668 ? $\alpha$  :=  $\sigma$ 
11669 ``
11670 Then, we need to solve:
11671 ``
11672 ?m ? $\alpha$  =?= StateT  $\sigma$  id  $\sigma$ 
11673 ==> instantiate metavar
11674 ?m  $\sigma$  =?= StateT  $\sigma$  id  $\sigma$ 
11675 ==> (approximate since it is a quasi-pattern unification constraint)
11676 ?m := fun  $\sigma$  => StateT  $\sigma$  id  $\sigma$ 
11677 ``
11678 Note that the constraint is not a Milner pattern because  $\sigma$  is in
11679 the local context of `?m`. We are ignoring the other possible solutions:
11680 ``
11681 ?m := fun  $\sigma'$  => StateT  $\sigma$  id  $\sigma$ 
11682 ?m := fun  $\sigma'$  => StateT  $\sigma'$  id  $\sigma$ 
11683 ?m := fun  $\sigma'$  => StateT  $\sigma$  id  $\sigma'$ 
11684 ``
11685
11686 We need the quasi-pattern approximation for elaborating recursor-like expressions (e.g., dependent `match with` expressions).
11687
11688 If we had use first-order unification, then we would have produced
11689 the right answer: `?m := StateT  $\sigma$  id`
11690
11691 Haskell would work on this example since it always uses
11692 first-order unification.
11693 -/
11694 def setElabConfig (cfg : Meta.Config) : Meta.Config :=
11695   { cfg with foApprox := true, ctxApprox := true, constApprox := false, quasiPatternApprox := false }
11696
11697 structure Context where
11698   fileName      : String
11699   fileMap       : FileMap
11700   declName?     : Option Name      := none
11701   macroStack    : MacroStack      := []
11702   currMacroScope : MacroScope     := firstFrontendMacroScope

```

```

11703 /- When `mayPostpone == true`, an elaboration function may interrupt its execution by throwing `Exception.postpone`.
11704 The function `elabTerm` catches this exception and creates fresh synthetic metavariable `?m`, stores `?m` in
11705 the list of pending synthetic metavariables, and returns `?m`. -/
11706 mayPostpone      : Bool           := true
11707 /- When `errToSorry` is set to true, the method `elabTerm` catches
11708 exceptions and converts them into synthetic `sorry`s.
11709 The implementation of choice nodes and overloaded symbols rely on the fact
11710 that when `errToSorry` is set to false for an elaboration function `F`, then
11711 `errToSorry` remains `false` for all elaboration functions invoked by `F`.
11712 That is, it is safe to transition `errToSorry` from `true` to `false`, but
11713 we must not set `errToSorry` to `true` when it is currently set to `false`. -/
11714 errToSorry       : Bool           := true
11715 /- When `autoBoundImplicit` is set to true, instead of producing
11716 an "unknown identifier" error for unbound variables, we generate an
11717 internal exception. This exception is caught at `elabBinders` and
11718 `elabTypeWithUnboldImplicit`. Both methods add implicit declarations
11719 for the unbound variable and try again. -/
11720 autoBoundImplicit : Bool           := false
11721 autoBoundImplicits : Std.PArray Expr := {}
11722 /-- Map from user name to internal unique name -/
11723 sectionVars      : NameMap Name   := {}
11724 /-- Map from internal name to fvar -/
11725 sectionFVars     : NameMap Expr   := {}
11726 /-- Enable/disable implicit lambdas feature. -/
11727 implicitLambda   : Bool           := true
11728
11729 /-- Saved context for postponed terms and tactics to be executed. -/
11730 structure SavedContext where
11731   declName? : Option Name
11732   options   : Options
11733   openDecls : List OpenDecl
11734   macroStack : MacroStack
11735   errToSorry : Bool
11736
11737 /-- We use synthetic metavariables as placeholders for pending elaboration steps. -/
11738 inductive SyntheticMVarKind where
11739   -- typeclass instance search
11740   | typeClass
11741   /- Similar to typeClass, but error messages are different.
11742   if `f?` is `some f`, we produce an application type mismatch error message.
11743   Otherwise, if `header?` is `some header`, we generate the error `(header ++ "has type" ++ eType ++ "but it is expected to have typ
11744   Otherwise, we generate the error `("type mismatch" ++ e ++ "has type" ++ eType ++ "but it is expected to have type" ++ expectedTyp
11745   | coe (header? : Option String) (eNew : Expr) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr)
11746   -- tactic block execution
11747   | tactic (tacticCode : Syntax) (ctx : SavedContext)
11748   -- `elabTerm` call that threw `Exception.postpone` (input is stored at `SyntheticMVarDecl.ref`)
11749   | postponed (ctx : SavedContext)

```

```

11750
11751 instance : ToString SyntheticMVarKind where
11752   toString
11753   | SyntheticMVarKind.typeClass    => "typeclass"
11754   | SyntheticMVarKind.coe ..       => "coe"
11755   | SyntheticMVarKind.tactic ..    => "tactic"
11756   | SyntheticMVarKind.postponed .. => "postponed"
11757
11758 structure SyntheticMVarDecl where
11759   mvarId : MVarId
11760   stx : Syntax
11761   kind : SyntheticMVarKind
11762
11763 inductive MVarErrorKind where
11764   | implicitArg (ctx : Expr)
11765   | hole
11766   | custom (msgData : MessageData)
11767
11768 instance : ToString MVarErrorKind where
11769   toString
11770   | MVarErrorKind.implicitArg ctx => "implicitArg"
11771   | MVarErrorKind.hole           => "hole"
11772   | MVarErrorKind.custom msg     => "custom"
11773
11774 structure MVarErrorInfo where
11775   mvarId      : MVarId
11776   ref         : Syntax
11777   kind        : MVarErrorKind
11778
11779 structure LetRecToLift where
11780   ref          : Syntax
11781   fvarId       : FVarId
11782   attrs        : Array Attribute
11783   shortDeclName : Name
11784   declName     : Name
11785   lctx         : LocalContext
11786   localInstances : LocalInstances
11787   type         : Expr
11788   val          : Expr
11789   mvarId       : MVarId
11790
11791 structure State where
11792   levelNames      : List Name      := []
11793   syntheticMVars  : List SyntheticMVarDecl := []
11794   mvarErrorInfos  : List MVarErrorInfo := []
11795   messages        : MessageLog := {}
11796   letRecsToLift   : List LetRecToLift := []

```

```

11797   infoState      : InfoState := {}
11798   deriving Inhabited
11799
11800 abbrev TermElabM := ReaderT Context $ StateRefT State MetaM
11801 abbrev TermElab  := Syntax → Option Expr → TermElabM Expr
11802
11803 open Meta
11804
11805 instance : Inhabited (TermElabM  $\alpha$ ) where
11806   default := throw arbitrary
11807
11808 structure SavedState where
11809   meta      : Meta.SavedState
11810   «elab»    : State
11811   deriving Inhabited
11812
11813 protected def saveState : TermElabM SavedState := do
11814   pure { meta := ( $\leftarrow$  Meta.saveState), «elab» := ( $\leftarrow$  get) }
11815
11816 def SavedState.restore (s : SavedState) : TermElabM Unit := do
11817   let traceState  $\leftarrow$  getTraceState -- We never backtrack trace message
11818   -- We also preserve `TacticInfo` nodes to be able to display the tactic state of broken tactic scripts
11819   let infoStateNew := ( $\leftarrow$  get).infoState
11820   let oldInfoSize  := s.elab.infoState.trees.size
11821   s.meta.restore
11822   set s.elab
11823   setTraceState traceState
11824   -- Add new `TacticInfo` nodes back to restored `infoState`
11825   modify fun s => { s with
11826     infoState.trees := infoStateNew.trees.foldl (init := s.infoState.trees) (start := oldInfoSize) fun trees info =>
11827       match info with
11828       | InfoTree.node (Info.ofTacticInfo _) _ => trees.push info
11829       | _ => trees
11830   }
11831
11832 instance : MonadBacktrack SavedState TermElabM where
11833   saveState      := Term.saveState
11834   restoreState b := b.restore
11835
11836 abbrev TermElabResult ( $\alpha$  : Type) := EStateM.Result Exception SavedState  $\alpha$ 
11837
11838 instance [Inhabited  $\alpha$ ] : Inhabited (TermElabResult  $\alpha$ ) where
11839   default := EStateM.Result.ok arbitrary arbitrary
11840
11841 def setMessageLog (messages : MessageLog) : TermElabM Unit :=
11842   modify fun s => { s with messages := messages }
11843

```

```

11844 def resetMessageLog : TermElabM Unit :=
11845   setMessageLog {}
11846
11847 def getMessageLog : TermElabM MessageLog :=
11848   return (← get).messages
11849
11850 /--
11851   Execute `x`, save resulting expression and new state.
11852   If `x` fails, then it also stores exception and new state.
11853   Remark: we do not capture `Exception.postpone`. -/
11854 @[inline] def observing (x : TermElabM  $\alpha$ ) : TermElabM (TermElabResult  $\alpha$ ) := do
11855   let s ← saveState
11856   try
11857     let e ← x
11858     let sNew ← saveState
11859     s.restore
11860     pure (EStateM.Result.ok e sNew)
11861   catch
11862   | ex@(Exception.error _ _) =>
11863     let sNew ← saveState
11864     s.restore
11865     pure (EStateM.Result.error ex sNew)
11866   | ex@(Exception.internal id _) =>
11867     if id == postponeExceptionId then s.restore
11868     throw ex
11869
11870 /--
11871   Apply the result/exception and state captured with `observing`.
11872   We use this method to implement overloaded notation and symbols. -/
11873 @[inline] def applyResult (result : TermElabResult  $\alpha$ ) : TermElabM  $\alpha$  :=
11874   match result with
11875   | EStateM.Result.ok a r      => do r.restore; pure a
11876   | EStateM.Result.error ex r => do r.restore; throw ex
11877
11878 /--
11879   Execute `x`, but keep state modifications only if `x` did not postpone.
11880   This method is useful to implement elaboration functions that cannot decide whether
11881   they need to postpone or not without updating the state. -/
11882 def commitIfDidNotPostpone (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
11883   -- We just reuse the implementation of `observing` and `applyResult`.
11884   let r ← observing x
11885   applyResult r
11886
11887 /--
11888   Execute `x` but discard changes performed at `Term.State` and `Meta.State`.
11889   Recall that the environment is at `Core.State`. Thus, any updates to it will
11890   be preserved. This method is useful for performing computations where all

```

```

11891  metavariable must be resolved or discarded. -/
11892 def withoutModifyingElabMetaState (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
11893   let s ← get
11894   let sMeta ← getThe Meta.State
11895   try
11896     x
11897   finally
11898     set s
11899     set sMeta
11900
11901 def getLevelNames : TermElabM (List Name) :=
11902   return (← get).levelNames
11903
11904 def getFVarLocalDecl! (fvar : Expr) : TermElabM LocalDecl := do
11905   match (← getLCtx).find? fvar.fvarId! with
11906   | some d => pure d
11907   | none   => unreachable!
11908
11909 instance : AddErrorMessageContext TermElabM where
11910   add ref msg := do
11911     let ctx ← read
11912     let ref := getBetterRef ref ctx.macroStack
11913     let msg ← addMessageContext msg
11914     let msg ← addMacroStack msg ctx.macroStack
11915     pure (ref, msg)
11916
11917 instance : MonadLog TermElabM where
11918   getRef      := getRef
11919   getFileMap  := return (← read).fileMap
11920   getFileName := return (← read).fileName
11921   logMessage msg := do
11922     let ctx ← readThe Core.Context
11923     let msg := { msg with data := MessageData.withNamingContext { currNamespace := ctx.currNamespace, openDecls := ctx.openDecls } msg }
11924     modify fun s => { s with messages := s.messages.add msg }
11925
11926 protected def getCurrMacroScope : TermElabM MacroScope := do pure (← read).currMacroScope
11927 protected def getMainModule      : TermElabM Name := do pure (← getEnv).mainModule
11928
11929 @[inline] protected def withFreshMacroScope (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
11930   let fresh ← modifyGetThe Core.State (fun st => (st.nextMacroScope, { st with nextMacroScope := st.nextMacroScope + 1 }))
11931   withReader (fun ctx => { ctx with currMacroScope := fresh }) x
11932
11933 instance : MonadQuotation TermElabM where
11934   getCurrMacroScope := Term.getCurrMacroScope
11935   getMainModule     := Term.getMainModule
11936   withFreshMacroScope := Term.withFreshMacroScope
11937

```

```

11938 instance : MonadInfoTree TermElabM where
11939   getInfoState      := return (← get).infoState
11940   modifyInfoState f := modify fun s => { s with infoState := f s.infoState }
11941
11942 unsafe def mkTermElabAttributeUnsafe : IO (KeyedDeclsAttribute TermElab) :=
11943   mkElabAttribute TermElab `Lean.Elab.Term.termElabAttribute `builtinTermElab `termElab `Lean.Parser.Term `Lean.Elab.Term.TermElab "termElab"
11944
11945 @[implementedBy mkTermElabAttributeUnsafe]
11946 constant mkTermElabAttribute : IO (KeyedDeclsAttribute TermElab)
11947
11948 builtin_initialize termElabAttribute : KeyedDeclsAttribute TermElab ← mkTermElabAttribute
11949
11950 /--
11951   Auxiliary datatype for presenting a Lean lvalue modifier.
11952   We represent a unelaborated lvalue as a `Syntax` (or `Expr`) and `List LVal`.
11953   Example: `a.foo[i].1` is represented as the `Syntax` `a` and the list
11954   `[LVal.fieldName "foo", LVal.getOp i, LVal.fieldIdx 1]`.
11955   Recall that the notation `a[i]` is not just for accessing arrays in Lean. -/
11956 inductive LVal where
11957   | fieldIdx (ref : Syntax) (i : Nat)
11958   /- Field `suffix?` is for producing better error messages because `x.y` may be a field access or a hierarchical/composite name.
11959   `ref` is the syntax object representing the field. `targetStx` is the target object being accessed. -/
11960   | fieldName (ref : Syntax) (name : String) (suffix? : Option Name) (targetStx : Syntax)
11961   | getOp      (ref : Syntax) (idx : Syntax)
11962
11963 def LVal.getRef : LVal → Syntax
11964 | LVal.fieldIdx ref _      => ref
11965 | LVal.fieldName ref ..   => ref
11966 | LVal.getOp ref _        => ref
11967
11968 def LVal.isFieldName : LVal → Bool
11969 | LVal.fieldName .. => true
11970 | _                 => false
11971
11972 instance : ToString LVal where
11973   toString
11974   | LVal.fieldIdx _ i      => toString i
11975   | LVal.fieldName _ n .. => n
11976   | LVal.getOp _ idx      => "[" ++ toString idx ++ "]"
11977
11978 def getDeclName? : TermElabM (Option Name) := return (← read).declName?
11979 def getLetRecsToLift : TermElabM (List LetRecToLift) := return (← get).letRecsToLift
11980 def isExprMVarAssigned (mvarId : MVarId) : TermElabM Bool := return (← getMCtx).isExprAssigned mvarId
11981 def getMVarDecl (mvarId : MVarId) : TermElabM MetavarDecl := return (← getMCtx).getDecl mvarId
11982 def assignLevelMVar (mvarId : MVarId) (val : Level) : TermElabM Unit := modifyThe Meta.State fun s => { s with mctx := s.mctx.assignLevel mvarId val }
11983
11984 def withDeclName (name : Name) (x : TermElabM α) : TermElabM α :=

```

```

11985   withReader (fun ctx => { ctx with declName? := name }) x
11986
11987 def setLevelNames (levelNames : List Name) : TermElabM Unit :=
11988   modify fun s => { s with levelNames := levelNames }
11989
11990 def withLevelNames (levelNames : List Name) (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
11991   let levelNamesSaved ← getLevelNames
11992   setLevelNames levelNames
11993   try x finally setLevelNames levelNamesSaved
11994
11995 def withoutErrToSorry (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  :=
11996   withReader (fun ctx => { ctx with errToSorry := false }) x
11997
11998 /- For testing `TermElabM` methods. The #eval command will sign the error. -/
11999 def throwErrorIfErrors : TermElabM Unit := do
12000   if (← get).messages.hasErrors then
12001     throwError "Error(s)"
12002
12003 @[inline] def traceAtCmdPos (cls : Name) (msg : Unit → MessageData) : TermElabM Unit :=
12004   withRef Syntax.missing $ trace cls msg
12005
12006 def ppGoal (mvarId : MVarId) : TermElabM Format :=
12007   Meta.ppGoal mvarId
12008
12009 open Level (LevelElabM)
12010
12011 def liftLevelM (x : LevelElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
12012   let ctx ← read
12013   let ref ← getRef
12014   let mctx ← getMCtx
12015   let ngen ← getNGen
12016   let lvlCtx : Level.Context := { options := (← getOptions), ref := ref, autoBoundImplicit := ctx.autoBoundImplicit }
12017   match (x lvlCtx).run { ngen := ngen, mctx := mctx, levelNames := (← getLevelNames) } with
12018   | EStateM.Result.ok a newS => setMCtx newS.mctx; setNGen newS.ngen; setLevelNames newS.levelNames; pure a
12019   | EStateM.Result.error ex _ => throw ex
12020
12021 def elabLevel (stx : Syntax) : TermElabM Level :=
12022   liftLevelM $ Level.elabLevel stx
12023
12024 /- Elaborate `x` with `stx` on the macro stack -/
12025 @[inline] def withMacroExpansion (beforeStx afterStx : Syntax) (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  :=
12026   withMacroExpansionInfo beforeStx afterStx do
12027     withReader (fun ctx => { ctx with macroStack := { before := beforeStx, after := afterStx } :: ctx.macroStack }) x
12028
12029 /-
12030   Add the given metavariable to the list of pending synthetic metavariables.
12031   The method `synthesizeSyntheticMVars` is used to process the metavariables on this list. -/

```



```

12032 def registerSyntheticMVar (stx : Syntax) (mvarId : MVarId) (kind : SyntheticMVarKind) : TermElabM Unit := do
12033   modify fun s => { s with syntheticMVars := { mvarId := mvarId, stx := stx, kind := kind } :: s.syntheticMVars }
12034
12035 def registerSyntheticMVarWithCurrRef (mvarId : MVarId) (kind : SyntheticMVarKind) : TermElabM Unit := do
12036   registerSyntheticMVar (← getRef) mvarId kind
12037
12038 def registerMVarErrorHoleInfo (mvarId : MVarId) (ref : Syntax) : TermElabM Unit := do
12039   modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.hole } :: s.mvarErrorInfos }
12040
12041 def registerMVarErrorImplicitArgInfo (mvarId : MVarId) (ref : Syntax) (app : Expr) : TermElabM Unit := do
12042   modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.implicitArg app } :: s.mvarErrorInfo
12043
12044 def registerMVarErrorCustomInfo (mvarId : MVarId) (ref : Syntax) (msgData : MessageData) : TermElabM Unit := do
12045   modify fun s => { s with mvarErrorInfos := { mvarId := mvarId, ref := ref, kind := MVarErrorKind.custom msgData } :: s.mvarErrorInfos
12046
12047 def registerCustomErrorIfMVar (e : Expr) (ref : Syntax) (msgData : MessageData) : TermElabM Unit :=
12048   match e.getAppFn with
12049   | Expr.mvar mvarId _ => registerMVarErrorCustomInfo mvarId ref msgData
12050   | _ => pure ()
12051
12052 /-
12053   Auxiliary method for reporting errors of the form "... contains metavariables ...".
12054   This kind of error is thrown, for example, at `Match.lean` where elaboration
12055   cannot continue if there are metavariables in patterns.
12056   We only want to log it if we haven't logged any error so far. -/
12057 def throwMVarError (m : MessageData) : TermElabM α := do
12058   if (← get).messages.hasErrors then
12059     throwAbortTerm
12060   else
12061     throwError m
12062
12063 def MVarErrorInfo.logError (mvarErrorInfo : MVarErrorInfo) (extraMsg? : Option MessageData) : TermElabM Unit := do
12064   match mvarErrorInfo.kind with
12065   | MVarErrorKind.implicitArg app => do
12066     let app ← instantiateMVars app
12067     let msg : MessageData := m!"don't know how to synthesize implicit argument{indentExpr app.setAppPPExplicitForExposingMVars}"
12068     let msg := msg ++ Format.line ++ "context:" ++ Format.line ++ MessageData.ofGoal mvarErrorInfo.mvarId
12069     logErrorAt mvarErrorInfo.ref (appendExtra msg)
12070   | MVarErrorKind.hole => do
12071     let msg : MessageData := "don't know how to synthesize placeholder"
12072     let msg := msg ++ Format.line ++ "context:" ++ Format.line ++ MessageData.ofGoal mvarErrorInfo.mvarId
12073     logErrorAt mvarErrorInfo.ref (MessageData.tagged `Elab.synthPlaceholder <| appendExtra msg)
12074   | MVarErrorKind.custom msg =>
12075     logErrorAt mvarErrorInfo.ref (appendExtra msg)
12076 where
12077   appendExtra (msg : MessageData) : MessageData :=
12078     match extraMsg? with

```

```

12079 | none => msg
12080 | some extraMsg => msg ++ extraMsg
12081
12082 /--
12083   Try to log errors for the unassigned metavariables `pendingMVarIds`.
12084
12085   Return `true` if there were "unfilled holes", and we should "abort" declaration.
12086   TODO: try to fill "all" holes using synthetic "sorry's"
12087
12088   Remark: We only log the "unfilled holes" as new errors if no error has been logged so far. -/
12089 def logUnassignedUsingErrorInfos (pendingMVarIds : Array MVarId) (extraMsg? : Option MessageData := none) : TermElabM Bool := do
12090   let s ← get
12091   let hasOtherErrors := s.messages.hasErrors
12092   let mut hasNewErrors := false
12093   let mut alreadyVisited : NameSet := {}
12094   for mvarErrorInfo in s.mvarErrorInfos do
12095     let mvarId := mvarErrorInfo.mvarId
12096     unless alreadyVisited.contains mvarId do
12097       alreadyVisited := alreadyVisited.insert mvarId
12098       let foundError ← withMVarContext mvarId do
12099         /- The metavariable `mvarErrorInfo.mvarId` may have been assigned or
12100            delayed assigned to another metavariable that is unassigned. -/
12101         let mvarDeps ← getMVars (mkMVar mvarId)
12102         if mvarDeps.any pendingMVarIds.contains then do
12103           unless hasOtherErrors do
12104             mvarErrorInfo.logError extraMsg?
12105             pure true
12106           else
12107             pure false
12108         if foundError then
12109           hasNewErrors := true
12110   return hasNewErrors
12111
12112 /-- Ensure metavariables registered using `registerMVarErrorInfos` (and used in the given declaration) have been assigned. -/
12113 def ensureNoUnassignedMVars (decl : Declaration) : TermElabM Unit := do
12114   let pendingMVarIds ← getMVarsAtDecl decl
12115   if (← logUnassignedUsingErrorInfos pendingMVarIds) then
12116     throwAbortCommand
12117
12118 /--
12119   Execute `x` without allowing it to postpone elaboration tasks.
12120   That is, `tryPostpone` is a noop. -/
12121 @[inline] def withoutPostponing (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  :=
12122   withReader (fun ctx => { ctx with mayPostpone := false }) x
12123
12124 /-- Creates syntax for `( `<ident> `: `<type> `)` -/
12125 def mkExplicitBinder (ident : Syntax) (type : Syntax) : Syntax :=

```

```

12126 mkNode ``Lean.Parser.Term.explicitBinder #[mkAtom "(", mkNullNode #[ident], mkNullNode #[mkAtom ":", type], mkNullNode, mkAtom ")"]
12127
12128 /--
12129   Convert unassigned universe level metavariables into parameters.
12130   The new parameter names are of the form `u_i` where `i >= nextParamIdx`.
12131   The method returns the updated expression and new `nextParamIdx`.
12132
12133   Remark: we make sure the generated parameter names do not clash with the universes at `ctx.levelNames`. -/
12134 def levelMVarToParam (e : Expr) (nextParamIdx : Nat := 1) : TermElabM (Expr × Nat) := do
12135   let mctx ← getMCtx
12136   let levelNames ← getLevelNames
12137   let r := mctx.levelMVarToParam (fun n => levelNames.elem n) e `u nextParamIdx
12138   setMCtx r.mctx
12139   pure (r.expr, r.nextParamIdx)
12140
12141 /-- Variant of `levelMVarToParam` where `nextParamIdx` is stored in a state monad. -/
12142 def levelMVarToParam' (e : Expr) : StateRefT Nat TermElabM Expr := do
12143   let nextParamIdx ← get
12144   let (e, nextParamIdx) ← levelMVarToParam e nextParamIdx
12145   set nextParamIdx
12146   pure e
12147
12148 /--
12149   Auxiliary method for creating fresh binder names.
12150   Do not confuse with the method for creating fresh free/meta variable ids. -/
12151 def mkFreshBinderName : TermElabM Name :=
12152   withFreshMacroScope $ MonadQuotation.addMacroScope `x
12153
12154 /--
12155   Auxiliary method for creating a `Syntax.ident` containing
12156   a fresh name. This method is intended for creating fresh binder names.
12157   It is just a thin layer on top of `mkFreshUserName`. -/
12158 def mkFreshIdent (ref : Syntax) : TermElabM Syntax :=
12159   return mkIdentFrom ref (← mkFreshBinderName)
12160
12161 private def applyAttributesCore
12162   (declName : Name) (attrs : Array Attribute)
12163   (applicationTime? : Option AttributeApplicationTime) : TermElabM Unit :=
12164   for attr in attrs do
12165     let env ← getEnv
12166     match getAttributeImpl env attr.name with
12167     | Except.error errMsg => throwError errMsg
12168     | Except.ok attrImpl =>
12169       match applicationTime? with
12170       | none => attrImpl.add declName attr.stx attr.kind
12171       | some applicationTime =>
12172         if applicationTime == attrImpl.applicationTime then

```

```

12173         attrImpl.add declName attr.stx attr.kind
12174
12175 /-- Apply given attributes **at** a given application time -/
12176 def applyAttributesAt (declName : Name) (attrs : Array Attribute) (applicationTime : AttributeApplicationTime) : TermElabM Unit :=
12177   applyAttributesCore declName attrs applicationTime
12178
12179 def applyAttributes (declName : Name) (attrs : Array Attribute) : TermElabM Unit :=
12180   applyAttributesCore declName attrs none
12181
12182 def mkTypeMismatchError (header? : Option String) (e : Expr) (eType : Expr) (expectedType : Expr) : TermElabM MessageData := do
12183   let header : MessageData := match header? with
12184     | some header => m! "{header} "
12185     | none       => m! "type mismatch{indentExpr e}\n"
12186   return m! "{header}{← mkHasTypeButIsExpectedMsg eType expectedType}"
12187
12188 def throwError (header? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr)
12189   (f? : Option Expr := none) (extraMsg? : Option MessageData := none) : TermElabM α := do
12190   /-
12191   We ignore `extraMsg?` for now. In all our tests, it contained no useful information. It was
12192   always of the form:
12193   ``
12194   failed to synthesize instance
12195   CoeT <eType> <e> <expectedType>
12196   ``
12197   We should revisit this decision in the future and decide whether it may contain useful information
12198   or not. -/
12199   let extraMsg := Format.nil
12200   /-
12201   let extraMsg : MessageData := match extraMsg? with
12202     | none       => Format.nil
12203     | some extraMsg => Format.line ++ extraMsg;
12204   -/
12205   match f? with
12206   | none => throwError "{← mkTypeMismatchError header? e eType expectedType}{extraMsg}"
12207   | some f => Meta.throwAppTypeMismatch f e extraMsg
12208
12209 @[inline] def withoutMacroStackAtErr (x : TermElabM α) : TermElabM α :=
12210   withTheReader Core.Context (fun (ctx : Core.Context) => { ctx with options := pp.macroStack.set ctx.options false }) x
12211
12212 /- Try to synthesize metavariable using type class resolution.
12213 This method assumes the local context and local instances of `instMVar` coincide
12214 with the current local context and local instances.
12215 Return `true` if the instance was synthesized successfully, and `false` if
12216 the instance contains unassigned metavariables that are blocking the type class
12217 resolution procedure. Throw an exception if resolution or assignment irrevocably fails. -/
12218 def synthesizeInstMVarCore (instMVar : MVarId) (maxResultSize? : Option Nat := none) : TermElabM Bool := do
12219   let instMVarDecl ← getMVarDecl instMVar

```

```

12220 let type := instMVarDecl.type
12221 let type ← instantiateMVars type
12222 let result ← trySynthInstance type maxResultSize?
12223 match result with
12224 | LOption.some val =>
12225   if (← isExprMVarAssigned instMVar) then
12226     let oldVal ← instantiateMVars (mkMVar instMVar)
12227     unless (← isDefEq oldVal val) do
12228       let oldValType ← inferType oldVal
12229       let valType ← inferType val
12230       unless (← isDefEq oldValType valType) do
12231         throwError "synthesized type class instance type is not definitionally equal to expected type, synthesized{indentExpr val}\nh.
12232         throwError "synthesized type class instance is not definitionally equal to expression inferred by typing rules, synthesized{ind
12233       else
12234         unless (← isDefEq (mkMVar instMVar) val) do
12235           throwError "failed to assign synthesized type class instance{indentExpr val}"
12236       pure true
12237 | LOption.undef    => pure false -- we will try later
12238 | LOption.none     => throwError "failed to synthesize instance{indentExpr type}"
12239
12240 register_builtin_option autoLift : Bool := {
12241   defValue := true
12242   descr   := "insert monadic lifts (i.e., `liftM` and `liftCoeM`) when needed"
12243 }
12244
12245 register_builtin_option maxCoeSize : Nat := {
12246   defValue := 16
12247   descr   := "maximum number of instances used to construct an automatic coercion"
12248 }
12249
12250 def synthesizeCoeInstMVarCore (instMVar : MVarId) : TermElabM Bool := do
12251   synthesizeInstMVarCore instMVar (some (maxCoeSize.get (← getOptions)))
12252
12253 /-
12254 The coercion from `α` to `Thunk α` cannot be implemented using an instance because it would
12255 eagerly evaluate `e` -/
12256 def tryCoeThunk? (expectedType : Expr) (eType : Expr) (e : Expr) : TermElabM (Option Expr) := do
12257   match expectedType with
12258   | Expr.app (Expr.const ``Thunk u _) arg _ =>
12259     if (← isDefEq eType arg) then
12260       pure (some (mkApp2 (mkConst ``Thunk.mk u) arg (mkSimpleThunk e)))
12261     else
12262       pure none
12263   | _ =>
12264     pure none
12265
12266 /-

```

```

12267 Try to apply coercion to make sure `e` has type `expectedType`.
12268 Relevant definitions:
12269 ```
12270 class CoeT ( $\alpha$  : Sort u) (a :  $\alpha$ ) ( $\beta$  : Sort v)
12271 abbrev coe { $\alpha$  : Sort u} { $\beta$  : Sort v} (a :  $\alpha$ ) [CoeT  $\alpha$  a  $\beta$ ] :  $\beta$ 
12272 ```
12273 -/
12274 private def tryCoe (errorMsgHeader? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr) : TermElabM Expr
12275   if ( $\leftarrow$  isDefEq expectedType eType) then
12276     return e
12277   else match ( $\leftarrow$  tryCoeThunk? expectedType eType e) with
12278   | some r => return r
12279   | none   =>
12280     let u  $\leftarrow$  getLevel eType
12281     let v  $\leftarrow$  getLevel expectedType
12282     let coeInstType := mkAppN (mkConst ``CoeT [u, v]) #[eType, e, expectedType]
12283     let mvar  $\leftarrow$  mkFreshExprMVar coeInstType MetavarKind.synthetic
12284     let eNew := mkAppN (mkConst ``coe [u, v]) #[eType, expectedType, e, mvar]
12285     let mvarId := mvar.mvarId!
12286     try
12287       withoutMacroStackAtErr do
12288         if ( $\leftarrow$  synthesizeCoeInstMVarCore mvarId) then
12289           expandCoe eNew
12290         else
12291           -- We create an auxiliary metavariable to represent the result, because we need to execute `expandCoe`
12292           -- after we synthesize `mvar`
12293           let mvarAux  $\leftarrow$  mkFreshExprMVar expectedType MetavarKind.syntheticOpaque
12294           registerSyntheticMVarWithCurrRef mvarAux.mvarId! (SyntheticMVarKind.coe errorMsgHeader? eNew expectedType eType e f?)
12295           return mvarAux
12296     catch
12297     | Exception.error _ msg => throwTypeMismatchError errorMsgHeader? expectedType eType e f? msg
12298     | _                     => throwTypeMismatchError errorMsgHeader? expectedType eType e f?
12299
12300 def isTypeApp? (type : Expr) : TermElabM (Option (Expr  $\times$  Expr)) := do
12301   let type  $\leftarrow$  withReducible $ whnf type
12302   match type with
12303   | Expr.app m  $\alpha$  _ => pure (some (( $\leftarrow$  instantiateMVars m), ( $\leftarrow$  instantiateMVars  $\alpha$ )))
12304   | _                => pure none
12305
12306 def synthesizeInst (type : Expr) : TermElabM Expr := do
12307   let type  $\leftarrow$  instantiateMVars type
12308   match ( $\leftarrow$  trySynthInstance type) with
12309   | LOption.some val => pure val
12310   | LOption.undef   => throwError "failed to synthesize instance{indentExpr type}"
12311   | LOption.none     => throwError "failed to synthesize instance{indentExpr type}"
12312
12313 def isMonadApp (type : Expr) : TermElabM Bool := do

```

```

12314 let some (m, _) ← isTypeApp? type | pure false
12315 return (← isMonad? m) |>.isSome
12316
12317 /--
12318 Try to coerce `a :  $\alpha$ ` into ` $m \beta$ ` by first coercing ` $a : \alpha$ ` into ` $\beta$ `, and then using `pure`.
12319 The method is only applied if ` $\alpha$ ` is not monadic (e.g., ` $\text{Nat} \rightarrow \text{IO Unit}$ `), and the head symbol
12320 of the resulting type is not a metavariable (e.g., ` $?m \text{Unit}$ ` or ` $\text{Bool} \rightarrow ?m \text{Nat}$ `).
12321
12322 The main limitation of the approach above is polymorphic code. As usual, coercions and polymorphism
12323 do not interact well. In the example above, the lift is successfully applied to `true`, `false` and `!y`
12324 since none of them is polymorphic
12325 ```
12326 def f (x : Bool) : IO Bool := do
12327   let y ← if x == 0 then IO.println "hello"; true else false;
12328   !y
12329   ```
12330 On the other hand, the following fails since `+` is polymorphic
12331 ```
12332 def f (x : Bool) : IO Nat := do
12333   IO.println x
12334   x + x -- Error: failed to synthesize `Add (IO Nat)`
12335   ```
12336 -/
12337 private def tryPureCoe? (errorMsgHeader? : Option String) (m  $\beta$   $\alpha$  a : Expr) : TermElabM (Option Expr) :=
12338   commitWhenSome? do
12339     let doIt : TermElabM (Option Expr) := do
12340       try
12341         let aNew ← tryCoe errorMsgHeader?  $\beta$   $\alpha$  a none
12342         let aNew ← mkPure m aNew
12343         pure (some aNew)
12344       catch _ =>
12345         pure none
12346     forallTelescope  $\alpha$  fun _  $\alpha$  => do
12347       if (← isMonadApp  $\alpha$ ) then
12348         pure none
12349       else if ! $\alpha$ .getAppFn.isMVar then
12350         doIt
12351       else
12352         pure none
12353
12354 /--
12355 Try coercions and monad lifts to make sure `e` has type `expectedType`.
12356
12357 If `expectedType` is of the form ` $n \beta$ `, we try monad lifts and other extensions.
12358 Otherwise, we just use the basic `tryCoe`.
12359
12360 Extensions for monads.

```

```

12361
12362 Given an expected type of the form `n β`, if `eType` is of the form `α`, but not `m α`
12363
12364 1 - Try to coerce `α` into `β`, and use `pure` to lift it to `n α`.
12365     It only works if `n` implements `Pure`
12366
12367 If `eType` is of the form `m α`. We use the following approaches.
12368
12369 1- Try to unify `n` and `m`. If it succeeds, then we use
12370     ```
12371     coeM {m : Type u → Type v} {α β : Type u} [∀ a, CoeT α a β] [Monad m] (x : m α) : m β
12372     ```
12373     `n` must be a `Monad` to use this one.
12374
12375 2- If there is monad lift from `m` to `n` and we can unify `α` and `β`, we use
12376     ```
12377     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
12378     ```
12379     Note that `n` may not be a `Monad` in this case. This happens quite a bit in code such as
12380     ```
12381     def g (x : Nat) : IO Nat := do
12382       IO.println x
12383       pure x
12384
12385     def f {m} [MonadLiftT IO m] : m Nat :=
12386       g 10
12387
12388     ```
12389
12390 3- If there is a monad lif from `m` to `n` and a coercion from `α` to `β`, we use
12391     ```
12392     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
12393     ```
12394
12395 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`.
12396 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
12397 we only have a coercion from decidable propositions. Approach 1 works because it constructs the coercion `CoeT (m Prop) (pure $ x > 0)`
12398 using the instance `pureCoeDepProp`.
12399
12400 Note that, approach 2 is more powerful than `tryCoe`.
12401 Recall that type class resolution never assigns metavariables created by other modules.
12402 Now, consider the following scenario
12403 ```lean
12404 def g (x : Nat) : IO Nat := ...
12405 def h (x : Nat) : StateT Nat IO Nat := do
12406   v ← g x;
12407   IO.Println v;

```



```

12408 ...
12409 ```
12410 Let's assume there is no other occurrence of `v` in `h`.
12411 Thus, we have that the expected of `g x` is `StateT Nat IO ?α`,
12412 and the given type is `IO Nat`. So, even if we add a coercion.
12413 ```
12414 instance {α m n} [MonadLiftT m n] {α} : Coe (m α) (n α) := ...
12415 ```
12416 It is not applicable because TC would have to assign `?α := Nat`.
12417 On the other hand, TC can easily solve `[MonadLiftT IO (StateT Nat IO)]`
12418 since this goal does not contain any metavariables. And then, we
12419 convert `g x` into `liftM $ g x`.
12420 -/
12421 private def tryLiftAndCoe (errorMsgHeader? : Option String) (expectedType : Expr) (eType : Expr) (e : Expr) (f? : Option Expr) : TermElabM Expr :=
12422   let expectedType ← instantiateMVars expectedType
12423   let eType ← instantiateMVars eType
12424   let throwMismatch {α} : TermElabM α := throwTypeMismatchError errorMsgHeader? expectedType eType e f?
12425   let tryCoeSimple : TermElabM Expr :=
12426     tryCoe errorMsgHeader? expectedType eType e f?
12427   let some (n, β) ← isTypeApp? expectedType | tryCoeSimple
12428   let tryPureCoeAndSimple : TermElabM Expr := do
12429     if autoLift.get (← getOptions) then
12430       match (← tryPureCoe? errorMsgHeader? n β eType e) with
12431       | some eNew => pure eNew
12432       | none      => tryCoeSimple
12433     else
12434       tryCoeSimple
12435   let some (m, α) ← isTypeApp? eType | tryPureCoeAndSimple
12436   if (← isDefEq m n) then
12437     let some monadInst ← isMonad? n | tryCoeSimple
12438     try expandCoe (← mkAppOptM ``coeM #[m, α, β, none, monadInst, e]) catch _ => throwMismatch
12439   else if autoLift.get (← getOptions) then
12440     try
12441       -- Construct lift from `m` to `n`
12442       let monadLiftType ← mkAppM ``MonadLiftT #[m, n]
12443       let monadLiftVal ← synthesizeInst monadLiftType
12444       let u_1 ← getDecLevel α
12445       let u_2 ← getDecLevel eType
12446       let u_3 ← getDecLevel expectedType
12447       let eNew := mkAppN (Lean.mkConst ``liftM [u_1, u_2, u_3]) #[m, n, monadLiftVal, α, e]
12448       let eNewType ← inferType eNew
12449       if (← isDefEq expectedType eNewType) then
12450         return eNew -- approach 2 worked
12451     else
12452       let some monadInst ← isMonad? n | tryCoeSimple
12453       let u ← getLevel α
12454       let v ← getLevel β

```

```

12455     let coeInstType := Lean.mkForall `a BinderInfo.default  $\alpha$  $ mkAppN (mkConst ``CoeT [u, v]) #[ $\alpha$ , mkBVar 0,  $\beta$ ]
12456     let coeInstVal ← synthesizeInst coeInstType
12457     let eNew ← expandCoe (← mkAppN (Lean.mkConst ``liftCoeM [u_1, u_2, u_3]) #[m, n,  $\alpha$ ,  $\beta$ , monadLiftVal, coeInstVal, monadInst, e])
12458     let eNewType ← inferType eNew
12459     unless (← isDefEq expectedType eNewType) do throwMismatch
12460     return eNew -- approach 3 worked
12461   catch _ =>
12462     /-
12463       If `m` is not a monad, then we try to use `tryPureCoe?` and then `tryCoe?`.
12464       Otherwise, we just try `tryCoe?`.
12465     -/
12466     match (← isMonad? m) with
12467     | none => tryPureCoeAndSimple
12468     | some _ => tryCoeSimple
12469   else
12470     tryCoeSimple
12471
12472 /--
12473   If `expectedType?` is `some t`, then ensure `t` and `eType` are definitionally equal.
12474   If they are not, then try coercions.
12475
12476   Argument `f?` is used only for generating error messages. -/
12477 def ensureHasTypeAux (expectedType? : Option Expr) (eType : Expr) (e : Expr)
12478   (f? : Option Expr := none) (errorMsgHeader? : Option String := none) : TermElabM Expr := do
12479   match expectedType? with
12480   | none => pure e
12481   | some expectedType =>
12482     if (← isDefEq eType expectedType) then
12483       pure e
12484     else
12485       tryLiftAndCoe errorMsgHeader? expectedType eType e f?
12486
12487 /--
12488   If `expectedType?` is `some t`, then ensure `t` and type of `e` are definitionally equal.
12489   If they are not, then try coercions. -/
12490 def ensureHasType (expectedType? : Option Expr) (e : Expr) (errorMsgHeader? : Option String := none) : TermElabM Expr :=
12491   match expectedType? with
12492   | none => pure e
12493   | _ => do
12494     let eType ← inferType e
12495     ensureHasTypeAux expectedType? eType e none errorMsgHeader?
12496
12497 private def mkSyntheticSorryFor (expectedType? : Option Expr) : TermElabM Expr := do
12498   let expectedType ← match expectedType? with
12499   | none => mkFreshTypeMVar
12500   | some expectedType => pure expectedType
12501   mkSyntheticSorry expectedType

```

```

12502
12503 private def exceptionToSorry (ex : Exception) (expectedType? : Option Expr) : TermElabM Expr := do
12504   let syntheticSorry ← mkSyntheticSorryFor expectedType?
12505   logException ex
12506   pure syntheticSorry
12507
12508 /-- If `mayPostpone == true`, throw `Exception.postpone`. -/
12509 def tryPostpone : TermElabM Unit := do
12510   if (← read).mayPostpone then
12511     throwPostpone
12512
12513 /-- If `mayPostpone == true` and `e`'s head is a metavariable, throw `Exception.postpone`. -/
12514 def tryPostponeIfMVar (e : Expr) : TermElabM Unit := do
12515   if e.getAppFn.isMVar then
12516     let e ← instantiateMVars e
12517     if e.getAppFn.isMVar then
12518       tryPostpone
12519
12520 def tryPostponeIfNoneOrMVar (e? : Option Expr) : TermElabM Unit :=
12521   match e? with
12522   | some e => tryPostponeIfMVar e
12523   | none   => tryPostpone
12524
12525 def tryPostponeIfHasMVars (expectedType? : Option Expr) (msg : String) : TermElabM Expr := do
12526   tryPostponeIfNoneOrMVar expectedType?
12527   let some expectedType ← pure expectedType? |
12528     throwError "{msg}, expected type must be known"
12529   let expectedType ← instantiateMVars expectedType
12530   if expectedType.hasExprMVar then
12531     tryPostpone
12532     throwError "{msg}, expected type contains metavariables{indentExpr expectedType}"
12533   pure expectedType
12534
12535 private def saveContext : TermElabM SavedContext :=
12536   return {
12537     macroStack := (← read).macroStack
12538     declName?  := (← read).declName?
12539     options    := (← getOptions)
12540     openDecls  := (← getOpenDecls)
12541     errToSorry := (← read).errToSorry
12542   }
12543
12544 def withSavedContext (savedCtx : SavedContext) (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
12545   withReader (fun ctx => { ctx with declName? := savedCtx.declName?, macroStack := savedCtx.macroStack, errToSorry := savedCtx.errToSor
12546     withTheReader Core.Context (fun ctx => { ctx with options := savedCtx.options, openDecls := savedCtx.openDecls })
12547     x
12548

```

```

12549 private def postponeElabTerm (stx : Syntax) (expectedType? : Option Expr) : TermElabM Expr := do
12550   trace[Elab.postpone] "{stx} : {expectedType?}"
12551   let mvar ← mkFreshExprMVar expectedType? MetavarKind.syntheticOpaque
12552   let ctx ← read
12553   registerSyntheticMVar stx mvar.mvarId! (SyntheticMVarKind.postponed (← saveContext))
12554   pure mvar
12555
12556 /-
12557   Helper function for `elabTerm` is tries the registered elaboration functions for `stxNode` kind until it finds one that supports the
12558   an error is found. -/
12559 private def elabUsingElabFnsAux (s : SavedState) (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone : Bool)
12560   : List TermElab → TermElabM Expr
12561 | [] => do throwError "unexpected syntax{indentD stx}"
12562 | (elabFn::elabFns) => do
12563   try
12564     elabFn stx expectedType?
12565   catch ex => match ex with
12566   | Exception.error _ =>
12567     if (← read).errToSorry then
12568       exceptionToSorry ex expectedType?
12569     else
12570       throw ex
12571   | Exception.internal id _ =>
12572     if (← read).errToSorry && id == abortTermExceptionId then
12573       exceptionToSorry ex expectedType?
12574     else if id == unsupportedSyntaxExceptionId then
12575       s.restore
12576       elabUsingElabFnsAux s stx expectedType? catchExPostpone elabFns
12577     else if catchExPostpone && id == postponeExceptionId then
12578       /- If `elab` threw `Exception.postpone`, we reset any state modifications.
12579       For example, we want to make sure pending synthetic metavariables created by `elab` before
12580       it threw `Exception.postpone` are discarded.
12581       Note that we are also discarding the messages created by `elab`.
12582
12583       For example, consider the expression.
12584       `((f.x a1).x a2).x a3`
12585       Now, suppose the elaboration of `f.x a1` produces an `Exception.postpone`.
12586       Then, a new metavariable `?m` is created. Then, `?m.x a2` also throws `Exception.postpone`
12587       because the type of `?m` is not yet known. Then another, metavariable `?n` is created, and
12588       finally `?n.x a3` also throws `Exception.postpone`. If we did not restore the state, we would
12589       keep "dead" metavariables `?m` and `?n` on the pending synthetic metavariable list. This is
12590       wasteful because when we resume the elaboration of `((f.x a1).x a2).x a3`, we start it from scratch
12591       and new metavariables are created for the nested functions. -/
12592       s.restore
12593       postponeElabTerm stx expectedType?
12594     else
12595       throw ex

```

```

12596
12597 private def elabUsingElabFns (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone : Bool) : TermElabM Expr := do
12598   let s ← saveState
12599   let table := termElabAttribute.ext.getState (← getEnv) |>.table
12600   let k := stx.getKind
12601   match table.find? k with
12602   | some elabFns => elabUsingElabFnsAux s stx expectedType? catchExPostpone elabFns
12603   | none         => throwError "elaboration function for '{k}' has not been implemented{indentD stx}"
12604
12605 instance : MonadMacroAdapter TermElabM where
12606   getCurrMacroScope := getCurrMacroScope
12607   getNextMacroScope := return (← getThe Core.State).nextMacroScope
12608   setNextMacroScope next := modifyThe Core.State fun s => { s with nextMacroScope := next }
12609
12610 private def isExplicit (stx : Syntax) : Bool :=
12611   match stx with
12612   | `(@$f) => true
12613   | _      => false
12614
12615 private def isExplicitApp (stx : Syntax) : Bool :=
12616   stx.getKind == ``Lean.Parser.Term.app && isExplicit stx[0]
12617
12618 /--
12619   Return true if `stx` if a lambda abstraction containing a `{}` or `[]` binder annotation.
12620   Example: `fun {α} (a : α) => a` -/
12621 private def isLambdaWithImplicit (stx : Syntax) : Bool :=
12622   match stx with
12623   | `(fun $binders* => $body) => binders.any fun b => b.isOfKind ``Lean.Parser.Term.implicitBinder || b.isOfKind `Lean.Parser.Term.inst
12624   | _                        => false
12625
12626 private partial def dropTermParens : Syntax → Syntax := fun stx =>
12627   match stx with
12628   | `(($stx)) => dropTermParens stx
12629   | _        => stx
12630
12631 private def isHole (stx : Syntax) : Bool :=
12632   match stx with
12633   | `(_)      => true
12634   | `(? _)    => true
12635   | `(? $x:ident) => true
12636   | _        => false
12637
12638 private def isTacticBlock (stx : Syntax) : Bool :=
12639   match stx with
12640   | `(by $x:tacticSeq) => true
12641   | _                  => false
12642

```

```

12643 private def isNoImplicitLambda (stx : Syntax) : Bool :=
12644   match stx with
12645   | `(noImplicitLambda% $x:term) => true
12646   | _ => false
12647
12648 def mkNoImplicitLambdaAnnotation (type : Expr) : Expr :=
12649   mkAnnotation `noImplicitLambda type
12650
12651 def hasNoImplicitLambdaAnnotation (type : Expr) : Bool :=
12652   annotation? `noImplicitLambda type |>.isSome
12653
12654 /-- Block usage of implicit lambdas if `stx` is `@f` or `@f arg1 ...` or `fun` with an implicit binder annotation. -/
12655 def blockImplicitLambda (stx : Syntax) : Bool :=
12656   let stx := dropTermParens stx
12657   -- TODO: make it extensible
12658   isExplicit stx || isExplicitApp stx || isLambdaWithImplicit stx || isHole stx || isTacticBlock stx || isNoImplicitLambda stx
12659
12660 /--
12661   Return normalized expected type if it is of the form `{a :  $\alpha$ }  $\rightarrow$   $\beta$ ` or `[a :  $\alpha$ ]  $\rightarrow$   $\beta$ ` and
12662   `blockImplicitLambda stx` is not true, else return `none`. -/
12663 private def useImplicitLambda? (stx : Syntax) (expectedType? : Option Expr) : TermElabM (Option Expr) :=
12664   if blockImplicitLambda stx then
12665     pure none
12666   else match expectedType? with
12667   | some expectedType => do
12668     if hasNoImplicitLambdaAnnotation expectedType then
12669       pure none
12670     else
12671       let expectedType ← whnfForall expectedType
12672       match expectedType with
12673       | Expr.forallE _ _ _ c => if c.binderInfo.isExplicit then pure none else pure $ some expectedType
12674       | _ => pure none
12675   | _ => pure none
12676
12677 private def elabImplicitLambdaAux (stx : Syntax) (catchExPostpone : Bool) (expectedType : Expr) (fvars : Array Expr) : TermElabM Expr :=
12678   let body ← elabUsingElabFns stx expectedType catchExPostpone
12679   let body ← ensureHasType expectedType body
12680   let r ← mkLambdaFVars fvars body
12681   trace[Elab.implicitForall] r
12682   pure r
12683
12684 private partial def elabImplicitLambda (stx : Syntax) (catchExPostpone : Bool) : Expr → Array Expr → TermElabM Expr
12685   | type@(Expr.forallE n d b c), fvars =>
12686     if c.binderInfo.isExplicit then
12687       elabImplicitLambdaAux stx catchExPostpone type fvars
12688     else withFreshMacroScope do
12689       let n ← MonadQuotation.addMacroScope n

```

```

12690     withLocalDecl n c.binderInfo d fun fvar => do
12691       let type ← whnfForall (b.instantiate1 fvar)
12692       elabImplicitLambda stx catchExPostpone type (fvars.push fvar)
12693   | type, fvars =>
12694     elabImplicitLambdaAux stx catchExPostpone type fvars
12695
12696 /- Main loop for `elabTerm` -/
12697 private partial def elabTermAux (expectedType? : Option Expr) (catchExPostpone : Bool) (implicitLambda : Bool) : Syntax → TermElabM Exp
12698   | Syntax.missing => mkSyntheticSorryFor expectedType?
12699   | stx => withFreshMacroScope <| withIncRecDepth do
12700     trace[Elab.step] "expected type: {expectedType?}, term\n{stx}"
12701     checkMaxHeartbeats "elaborator"
12702     withNestedTraces do
12703       let env ← getEnv
12704       let stxNew? ← catchInternalId unsupportedSyntaxExceptionId
12705         (do let newStx ← adaptMacro (getMacros env) stx; pure (some newStx))
12706         (fun _ => pure none)
12707       match stxNew? with
12708       | some stxNew => withMacroExpansion stx stxNew $ elabTermAux expectedType? catchExPostpone implicitLambda stxNew
12709       | _ =>
12710         let implicit? ← if implicitLambda && (← read).implicitLambda then useImplicitLambda? stx expectedType? else pure none
12711         match implicit? with
12712         | some expectedType => elabImplicitLambda stx catchExPostpone expectedType #[]
12713         | none => elabUsingElabFns stx expectedType? catchExPostpone
12714
12715 def addTermInfo (stx : Syntax) (e : Expr) : TermElabM Unit := do
12716   if (← getInfoState).enabled then
12717     pushInfoLeaf <| Info.ofTermInfo { lctx := (← getLCtx), expr := e, stx := stx }
12718
12719 def getSyntheticMVarDecl? (mvarId : MVarId) : TermElabM (Option SyntheticMVarDecl) :=
12720   return (← get).syntheticMVars.find? fun d => d.mvarId == mvarId
12721
12722 def mkTermInfo (stx : Syntax) (e : Expr) : TermElabM (Sum Info MVarId) := do
12723   let isHole? : TermElabM (Option MVarId) := do
12724     match e with
12725     | Expr.mvar mvarId _ =>
12726       match (← getSyntheticMVarDecl? mvarId) with
12727       | some { kind := SyntheticMVarKind.tactic .., .. } => return mvarId
12728       | some { kind := SyntheticMVarKind.postponed .., .. } => return mvarId
12729       | _ => return none
12730     | _ => pure none
12731   match (← isHole?) with
12732   | none => return Sum.inl <| Info.ofTermInfo { lctx := (← getLCtx), expr := e, stx := stx }
12733   | some mvarId => return Sum.inr mvarId
12734
12735 /- Store in the `InfoTree` that `e` is a "dot"-completion target. -/
12736 def addDotCompletionInfo (stx : Syntax) (e : Expr) (expectedType? : Option Expr) (field? : Option Syntax := none) : TermElabM Unit := d

```

```

12737   addCompletionInfo <| CompletionInfo.dot { expr := e, stx := stx, lctx := (← getLCtx) } (field? := field?) (expectedType? := expectedT
12738
12739 /--
12740   Main function for elaborating terms.
12741   It extracts the elaboration methods from the environment using the node kind.
12742   Recall that the environment has a mapping from `SyntaxNodeKind` to `TermElab` methods.
12743   It creates a fresh macro scope for executing the elaboration method.
12744   All unlogged trace messages produced by the elaboration method are logged using
12745   the position information at `stx`. If the elaboration method throws an `Exception.error` and `errToSorry == true`,
12746   the error is logged and a synthetic sorry expression is returned.
12747   If the elaboration throws `Exception.postpone` and `catchExPostpone == true`,
12748   a new synthetic metavariable of kind `SyntheticMVarKind.postponed` is created, registered,
12749   and returned.
12750   The option `catchExPostpone == false` is used to implement `resumeElabTerm`
12751   to prevent the creation of another synthetic metavariable when resuming the elaboration.
12752
12753   If `implicitLambda == true`, then disable implicit lambdas feature for the given syntax, but not for its subterms.
12754   We use this flag to implement, for example, the `@` modifier. If `Context.implicitLambda == false`, then this parameter has no effect
12755   -/
12756 def elabTerm (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone := true) (implicitLambda := true) : TermElabM Expr :=
12757   withInfoContext' (withRef stx <| elabTermAux expectedType? catchExPostpone implicitLambda stx) (mkTermInfo stx)
12758
12759 def elabTermEnsuringType (stx : Syntax) (expectedType? : Option Expr) (catchExPostpone := true) (implicitLambda := true) (errorMsgHeade
12760   let e ← elabTerm stx expectedType? catchExPostpone implicitLambda
12761   withRef stx <| ensureHasType expectedType? e errorMsgHeader?
12762
12763 /-- Adapt a syntax transformation to a regular, term-producing elaborator. -/
12764 def adaptExpander (exp : Syntax → TermElabM Syntax) : TermElab := fun stx expectedType? => do
12765   let stx' ← exp stx
12766   withMacroExpansion stx stx' $ elabTerm stx' expectedType?
12767
12768 def mkInstMVar (type : Expr) : TermElabM Expr := do
12769   let mvar ← mkFreshExprMVar type MetavarKind.synthetic
12770   let mvarId := mvar.mvarId!
12771   unless (← synthesizeInstMVarCore mvarId) do
12772     registerSyntheticMVarWithCurrRef mvarId SyntheticMVarKind.typeClass
12773   pure mvar
12774
12775 /--
12776   Relevant definitions:
12777   ```
12778   class CoeSort (α : Sort u) (β : outParam (Sort v))
12779   abbrev coeSort {α : Sort u} {β : Sort v} (a : α) [CoeSort α β] : β
12780   ```
12781   -/
12782 private def tryCoeSort (α : Expr) (a : Expr) : TermElabM Expr := do
12783   let β ← mkFreshTypeMVar

```



```

12784 let u ← getLevel  $\alpha$ 
12785 let v ← getLevel  $\beta$ 
12786 let coeSortInstType := mkAppN (Lean.mkConst ``CoeSort [u, v]) #[ $\alpha$ ,  $\beta$ ]
12787 let mvar ← mkFreshExprMVar coeSortInstType MetavarKind.synthetic
12788 let mvarId := mvar.mvarId!
12789 try
12790   withoutMacroStackAtErr do
12791     if (← synthesizeCoeInstMVarCore mvarId) then
12792       expandCoe <| mkAppN (Lean.mkConst ``coeSort [u, v]) #[ $\alpha$ ,  $\beta$ , a, mvar]
12793     else
12794       throwError "type expected"
12795 catch
12796   | Exception.error _ msg => throwError "type expected\n{msg}"
12797   | _                     => throwError "type expected"
12798
12799 /--
12800   Make sure `e` is a type by inferring its type and making sure it is a `Expr.sort`
12801   or is unifiable with `Expr.sort`, or can be coerced into one. -/
12802 def ensureType (e : Expr) : TermElabM Expr := do
12803   if (← isType e) then
12804     pure e
12805   else
12806     let eType ← inferType e
12807     let u ← mkFreshLevelMVar
12808     if (← isDefEq eType (mkSort u)) then
12809       pure e
12810     else
12811       tryCoeSort eType e
12812
12813 /-- Elaborate `stx` and ensure result is a type. -/
12814 def elabType (stx : Syntax) : TermElabM Expr := do
12815   let u ← mkFreshLevelMVar
12816   let type ← elabTerm stx (mkSort u)
12817   withRef stx $ ensureType type
12818
12819 /--
12820   Enable auto-bound implicits, and execute `k` while catching auto bound implicit exceptions. When an exception is caught,
12821   a new local declaration is created, registered, and `k` is tried to be executed again. -/
12822 partial def withAutoBoundImplicit (k : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
12823   let flag := autoBoundImplicitLocal.get (← getOptions)
12824   if flag then
12825     withReader (fun ctx => { ctx with autoBoundImplicit := flag, autoBoundImplicits := {} }) do
12826       let rec loop (s : SavedState) : TermElabM  $\alpha$  := do
12827         try
12828           k
12829         catch
12830           | ex => match isAutoBoundImplicitLocalException? ex with

```

```

12831         | some n =>
12832         -- Restore state, declare `n`, and try again
12833         s.restore
12834         withLocalDecl n BinderInfo.implicit (← mkFreshTypeMVar) fun x =>
12835             withReader (fun ctx => { ctx with autoBoundImplicits := ctx.autoBoundImplicits.push x } ) do
12836                 loop (← saveState)
12837         | none => throw ex
12838     loop (← saveState)
12839 else
12840     k
12841
12842 def withoutAutoBoundImplicit (k : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
12843     withReader (fun ctx => { ctx with autoBoundImplicit := false, autoBoundImplicits := {} }) k
12844
12845 /-
12846     Return `autoBoundImplicits ++ xs`.
12847     This methoid throws an error if a variable in `autoBoundImplicits` depends on some `x` in `xs` -/
12848 def addAutoBoundImplicits (xs : Array Expr) : TermElabM (Array Expr) := do
12849     let autoBoundImplicits := (← read).autoBoundImplicits
12850     for auto in autoBoundImplicits do
12851         let localDecl ← getLocalDecl auto.fvarId!
12852         for x in xs do
12853             if (← getMCtx).localDeclDependsOn localDecl x.fvarId! then
12854                 throwError "invalid auto implicit argument '{auto}', it depends on explicitly provided argument '{x}'"
12855     return autoBoundImplicits.toArray ++ xs
12856
12857 def mkAuxName (suffix : Name) : TermElabM Name := do
12858     match (← read).declName? with
12859     | none => throwError "auxiliary declaration cannot be created when declaration name is not available"
12860     | some declName => Lean.mkAuxName (declName ++ suffix) 1
12861
12862 builtin_initialize registerTraceClass `Elab.letrec
12863
12864 /- Return true if mvarId is an auxiliary metavariable created for compiling `let rec` or it
12865     is delayed assigned to one. -/
12866 def isLetRecAuxMVar (mvarId : MVarId) : TermElabM Bool := do
12867     trace[Elab.letrec] "mvarId: {mkMVar mvarId} letrecMVars: {(← get).letRecsToLift.map (mkMVar $ ·.mvarId)}"
12868     let mvarId := (← getMCtx).getDelayedRoot mvarId
12869     trace[Elab.letrec] "mvarId root: {mkMVar mvarId}"
12870     return (← get).letRecsToLift.any (·.mvarId == mvarId)
12871
12872 /- =====
12873     Builtin elaboration functions
12874     ===== -/
12875
12876 @[builtinTermElab «prop»] def elabProp : TermElab := fun _ _ =>
12877     return mkSort levelZero

```

```

12878
12879 private def elabOptLevel (stx : Syntax) : TermElabM Level :=
12880   if stx.isNone then
12881     pure levelZero
12882   else
12883     elabLevel stx[0]
12884
12885 @[builtinTermElab «sort»] def elabSort : TermElab := fun stx _ =>
12886   return mkSort (← elabOptLevel stx[1])
12887
12888 @[builtinTermElab «type»] def elabTypeStx : TermElab := fun stx _ =>
12889   return mkSort (mkLevelSucc (← elabOptLevel stx[1]))
12890
12891 /-
12892   the method `resolveName` adds a completion point for it using the given
12893   expected type. Thus, we propagate the expected type if `stx[0]` is an identifier.
12894   It doesn't "hurt" if the identifier can be resolved because the expected type is not used in this case.
12895   Recall that if the name resolution fails a synthetic sorry is returned.-/
12896
12897 @[builtinTermElab «pipeCompletion»] def elabPipeCompletion : TermElab := fun stx expectedType? => do
12898   let e ← elabTerm stx[0] none
12899   unless e.isSorry do
12900     addDotCompletionInfo stx e expectedType?
12901   throwErrorAt stx[1] "invalid field notation, identifier or numeral expected"
12902
12903 @[builtinTermElab «completion»] def elabCompletion : TermElab := fun stx expectedType? => do
12904   /- `ident.` is ambiguous in Lean, we may try to be completing a declaration name or access a "field". -/
12905   if stx[0].isIdent then
12906     /- If we can elaborate the identifier successfully, we assume it a dot-completion. Otherwise, we treat it as
12907         identifier completion with a dangling `.`.
12908         Recall that the server falls back to identifier completion when dot-completion fails. -/
12909     let s ← saveState
12910     try
12911       let e ← elabTerm stx[0] none
12912       addDotCompletionInfo stx e expectedType?
12913     catch _ =>
12914       s.restore
12915       addCompletionInfo <| CompletionInfo.id stx stx[0].getId (danglingDot := true) (← getLCtx) expectedType?
12916       throwErrorAt stx[1] "invalid field notation, identifier or numeral expected"
12917   else
12918     elabPipeCompletion stx expectedType?
12919
12920 @[builtinTermElab «hole»] def elabHole : TermElab := fun stx expectedType? => do
12921   let mvar ← mkFreshExprMVar expectedType?
12922   registerMVarErrorHoleInfo mvar.mvarId! stx
12923   pure mvar
12924

```

```

12925 @[builtinTermElab «syntheticHole»] def elabSyntheticHole : TermElab := fun stx expectedType? => do
12926   let arg := stx[1]
12927   let userName := if arg.isIdent then arg.getId else Name.anonymous
12928   let mkNewHole : Unit → TermElabM Expr := fun _ => do
12929     let mvar ← mkFreshExprMVar expectedType? MetavarKind.syntheticOpaque userName
12930     registerMVarErrorHoleInfo mvar.mvarId! stx
12931     pure mvar
12932   if userName.isAnonymous then
12933     mkNewHole ()
12934   else
12935     let mctx ← getMCtx
12936     match mctx.findUserName? userName with
12937     | none => mkNewHole ()
12938     | some mvarId =>
12939       let mvar := mkMVar mvarId
12940       let mvarDecl ← getMVarDecl mvarId
12941       let lctx ← getLCtx
12942       if mvarDecl.lctx.isSubPrefixOf lctx then
12943         pure mvar
12944       else match mctx.getExprAssignment? mvarId with
12945       | some val =>
12946         let val ← instantiateMVars val
12947         if mctx.isWellFormed lctx val then
12948           pure val
12949         else
12950           withLCtx mvarDecl.lctx mvarDecl.localInstances do
12951             throwError "synthetic hole has already been defined and assigned to value incompatible with the current context{indentExpr
12952       | none =>
12953         if mctx.isDelayedAssigned mvarId then
12954           -- We can try to improve this case if needed.
12955           throwError "synthetic hole has already beend defined and delayed assigned with an incompatible local context"
12956         else if lctx.isSubPrefixOf mvarDecl.lctx then
12957           let mvarNew ← mkNewHole ()
12958           modifyMCtx fun mctx => mctx.assignExpr mvarId mvarNew
12959           pure mvarNew
12960         else
12961           throwError "synthetic hole has already been defined with an incompatible local context"
12962
12963 private def mkTacticMVar (type : Expr) (tacticCode : Syntax) : TermElabM Expr := do
12964   let mvar ← mkFreshExprMVar type MetavarKind.syntheticOpaque
12965   let mvarId := mvar.mvarId!
12966   let ref ← getRef
12967   let declName? ← getDeclName?
12968   registerSyntheticMVar ref mvarId <| SyntheticMVarKind.tactic tacticCode (← saveContext)
12969   return mvar
12970
12971 @[builtinTermElab byTactic] def elabByTactic : TermElab := fun stx expectedType? =>

```

```

12972 match expectedType? with
12973 | some expectedType => mkTacticMVar expectedType stx
12974 | none => throwError ("invalid 'by' tactic, expected type has not been provided")
12975
12976 @[builtinTermElab noImplicitLambda] def elabNoImplicitLambda : TermElab := fun stx expectedType? =>
12977   elabTerm stx[1] (mkNoImplicitLambdaAnnotation <$> expectedType?)
12978
12979 def resolveLocalName (n : Name) : TermElabM (Option (Expr × List String)) := do
12980   let lctx ← getLCtx
12981   let view := extractMacroScopes n
12982   let rec loop (n : Name) (projs : List String) :=
12983     match lctx.findFromUserName? { view with name := n }.review with
12984     | some decl => some (decl.toExpr, projs)
12985     | none      => match n with
12986     | Name.str pre s _ => loop pre (s::projs)
12987     | _                => none
12988   return loop view.name []
12989
12990 /- Return true iff `stx` is a `Syntax.ident`, and it is a local variable. -/
12991 def isLocalIdent? (stx : Syntax) : TermElabM (Option Expr) :=
12992   match stx with
12993   | Syntax.ident _ _ val _ => do
12994     let r? ← resolveLocalName val
12995     match r? with
12996     | some (fvar, []) => pure (some fvar)
12997     | _               => pure none
12998   | _ => pure none
12999
13000 /--
13001   Create an `Expr.const` using the given name and explicit levels.
13002   Remark: fresh universe metavariables are created if the constant has more universe
13003   parameters than `explicitLevels`. -/
13004 def mkConst (constName : Name) (explicitLevels : List Level := []) : TermElabM Expr := do
13005   let cinfo ← getConstInfo constName
13006   if explicitLevels.length > cinfo.levelParams.length then
13007     throwError "too many explicit universe levels"
13008   else
13009     let numMissingLevels := cinfo.levelParams.length - explicitLevels.length
13010     let us ← mkFreshLevelMVars numMissingLevels
13011     pure $ Lean.mkConst constName (explicitLevels ++ us)
13012
13013 private def mkConsts (candidates : List (Name × List String)) (explicitLevels : List Level) : TermElabM (List (Expr × List String)) :=
13014   candidates.foldlM (init := []) fun result (constName, projs) => do
13015     -- TODO: better support for `mkConst` failure. We may want to cache the failures, and report them if all candidates fail.
13016     let const ← mkConst constName explicitLevels
13017     return (const, projs) :: result
13018

```

```

13019 def resolveName (stx : Syntax) (n : Name) (preresolved : List (Name × List String)) (explicitLevels : List Level) (expectedType? : Opti
13020   try
13021     if let some (e, projs) ← resolveLocalName n then
13022       unless explicitLevels.isEmpty do
13023         throwError "invalid use of explicit universe parameters, '{e}' is a local"
13024       return [(e, projs)]
13025     -- check for section variable capture by a quotation
13026     let ctx ← read
13027     if let some (e, projs) := preresolved.findSome? fun (n, projs) => ctx.sectionFVars.find? n |>.map (·, projs) then
13028       return [(e, projs)] -- section variables should shadow global decls
13029     if preresolved.isEmpty then
13030       process (← resolveGlobalName n)
13031     else
13032       process preresolved
13033   catch ex =>
13034     if preresolved.isEmpty && explicitLevels.isEmpty then
13035       addCompletionInfo <| CompletionInfo.id stx stx.getId (danglingDot := false) (← getLCtx) expectedType?
13036     throw ex
13037 where process (candidates : List (Name × List String)) : TermElabM (List (Expr × List String)) := do
13038   if candidates.isEmpty then
13039     if (← read).autoBoundImplicit && isValidAutoBoundImplicitName n then
13040       throwAutoBoundImplicitLocal n
13041     else
13042       throwError "unknown identifier '{Lean.mkConst n}'"
13043   if preresolved.isEmpty && explicitLevels.isEmpty then
13044     addCompletionInfo <| CompletionInfo.id stx stx.getId (danglingDot := false) (← getLCtx) expectedType?
13045   mkConsts candidates explicitLevels
13046
13047 /--
13048   Similar to `resolveName`, but creates identifiers for the main part and each projection with position information derived from `ident
13049   Example: Assume resolveName `v.head.bla.boo` produces `(v.head, ["bla", "boo"])`, then this method produces
13050   `(v.head, id, [f₁, f₂])` where `id` is an identifier for `v.head`, and `f₁` and `f₂` are identifiers for fields `"bla"` and `"boo"`.
13051 def resolveName' (ident : Syntax) (explicitLevels : List Level) (expectedType? : Option Expr := none) : TermElabM (List (Expr × Syntax :
13052   match ident with
13053   | Syntax.ident info rawStr n preresolved =>
13054     let r ← resolveName ident n preresolved explicitLevels expectedType?
13055     r.mapM fun (c, fields) => do
13056       let (cSstr, fields) := fields.foldr (init := (rawStr, [])) fun field (restSstr, fs) =>
13057         let fieldSstr := restSstr.takeRightWhile (· ≠ '.')
13058         ({ restSstr with stopPos := restSstr.stopPos - (fieldSstr.bsize + 1) }, (field, fieldSstr) :: fs)
13059       let mkIdentFromPos pos rawVal val :=
13060         let info := match info with
13061         | SourceInfo.original .. => SourceInfo.original "".toSubstring pos "".toSubstring
13062         | _ => SourceInfo.synthetic pos (pos + rawVal.bsize)
13063         Syntax.ident info rawVal val []
13064       let id := match c with
13065       | Expr.const id _ _ => id

```

```

13066 | Expr.fvar id _ => id
13067 | _ => unreachable!
13068 let id := mkIdentFromPos (ident.getPos?.getD 0) cSstr id
13069 match info.getPos? with
13070 | none =>
13071   return (c, id, fields.map fun (field, _) => mkIdentFrom ident (Name.mkSimple field))
13072 | some pos =>
13073   let mut pos := pos + cSstr.bsize + 1
13074   let mut newFields := #[]
13075   for (field, fieldSstr) in fields do
13076     newFields := newFields.push <| mkIdentFromPos pos fieldSstr (Name.mkSimple field)
13077     pos := pos + fieldSstr.bsize + 1
13078   return (c, id, newFields.toList)
13079 | _ => throwError "identifier expected"
13080
13081 def resolveId? (stx : Syntax) (kind := "term") : TermElabM (Option Expr) :=
13082   match stx with
13083   | Syntax.ident _ _ val preresolved => do
13084     let rs ← try resolveName stx val preresolved [] catch _ => pure []
13085     let rs := rs.filter fun (f, projs) => projs.isEmpty
13086     let fs := rs.map fun (f, _) => f
13087     match fs with
13088     | [] => pure none
13089     | [f] => pure (some f)
13090     | _ => throwError "ambiguous {kind}, use fully qualified name, possible interpretations {fs}"
13091   | _ => throwError "identifier expected"
13092
13093 @[builtinTermElab cdot] def elabBadCDot : TermElab := fun stx _ =>
13094   throwError "invalid occurrence of `` notation, it must be surrounded by parentheses (e.g. ``(· + 1)`)"
13095
13096 @[builtinTermElab strLit] def elabStrLit : TermElab := fun stx _ => do
13097   match stx.isStrLit? with
13098   | some val => pure $ mkStrLit val
13099   | none => throwIllFormedSyntax
13100
13101 private def mkFreshTypeMVarFor (expectedType? : Option Expr) : TermElabM Expr := do
13102   let typeMVar ← mkFreshTypeMVar MetavarKind.synthetic
13103   match expectedType? with
13104   | some expectedType => discard <| isDefEq expectedType typeMVar
13105   | _ => pure ()
13106   return typeMVar
13107
13108 @[builtinTermElab numLit] def elabNumLit : TermElab := fun stx expectedType? => do
13109   let val ← match stx.isNatLit? with
13110   | some val => pure val
13111   | none => throwIllFormedSyntax
13112   let typeMVar ← mkFreshTypeMVarFor expectedType?

```

```

13113 let u ← getDecLevel typeMVar
13114 let mvar ← mkInstMVar (mkApp2 (Lean.mkConst ``OfNat [u]) typeMVar (mkNatLit val))
13115 let r := mkApp3 (Lean.mkConst ``OfNat.ofNat [u]) typeMVar (mkNatLit val) mvar
13116 registerMVarErrorImplicitArgInfo mvar.mvarId! stx r
13117 return r
13118
13119 @[builtinTermElab rawNatLit] def elabRawNatLit : TermElab := fun stx expectedType? => do
13120   match stx[1].isNatLit? with
13121   | some val => return mkNatLit val
13122   | none     => throwIllFormedSyntax
13123
13124 @[builtinTermElab scientificLit]
13125 def elabScientificLit : TermElab := fun stx expectedType? => do
13126   match stx.isScientificLit? with
13127   | none     => throwIllFormedSyntax
13128   | some (m, sign, e) =>
13129     let typeMVar ← mkFreshTypeMVarFor expectedType?
13130     let u ← getDecLevel typeMVar
13131     let mvar ← mkInstMVar (mkApp (Lean.mkConst ``OfScientific [u]) typeMVar)
13132     return mkApp5 (Lean.mkConst ``OfScientific.ofScientific [u]) typeMVar mvar (mkNatLit m) (toExpr sign) (mkNatLit e)
13133
13134 @[builtinTermElab charLit] def elabCharLit : TermElab := fun stx _ => do
13135   match stx.isCharLit? with
13136   | some val => return mkApp (Lean.mkConst ``Char.ofNat) (mkNatLit val.toNat)
13137   | none     => throwIllFormedSyntax
13138
13139 @[builtinTermElab quotedName] def elabQuotedName : TermElab := fun stx _ =>
13140   match stx[0].isNameLit? with
13141   | some val => pure $ toExpr val
13142   | none     => throwIllFormedSyntax
13143
13144 @[builtinTermElab doubleQuotedName] def elabDoubleQuotedName : TermElab := fun stx _ => do
13145   match stx[1].isNameLit? with
13146   | some val => toExpr (← resolveGlobalConstNoOverloadWithInfo stx[1] val)
13147   | none     => throwIllFormedSyntax
13148
13149 @[builtinTermElab typeOf] def elabTypeOf : TermElab := fun stx _ => do
13150   inferType (← elabTerm stx[1] none)
13151
13152 @[builtinTermElab ensureTypeOf] def elabEnsureTypeOf : TermElab := fun stx expectedType? =>
13153   match stx[2].isStrLit? with
13154   | none     => throwIllFormedSyntax
13155   | some msg => do
13156     let refTerm ← elabTerm stx[1] none
13157     let refTermType ← inferType refTerm
13158     elabTermEnsuringType stx[3] refTermType (errorMsgHeader? := msg)
13159

```



```

13160 @[builtinTermElab ensureExpectedType] def elabEnsureExpectedType : TermElab := fun stx expectedType? =>
13161   match stx[1].isStrLit? with
13162   | none      => throwIllFormedSyntax
13163   | some msg => elabTermEnsuringType stx[2] expectedType? (errorMsgHeader? := msg)
13164
13165 @[builtinTermElab «open»] def elabOpen : TermElab := fun stx expectedType? => do
13166   try
13167     pushScope
13168     let openDecls ← elabOpenDecl stx[1]
13169     withTheReader Core.Context (fun ctx => { ctx with openDecls := openDecls }) do
13170       elabTerm stx[3] expectedType?
13171   finally
13172     popScope
13173
13174 @[builtinTermElab «set_option»] def elabSetOption : TermElab := fun stx expectedType? => do
13175   let options ← Elab.elabSetOption stx[1] stx[2]
13176   withTheReader Core.Context (fun ctx => { ctx with maxRecDepth := maxRecDepth.get options, options := options }) do
13177     elabTerm stx[4] expectedType?
13178
13179 private def mkSomeContext : Context := {
13180   fileName      := "<TermElabM>"
13181   fileMap       := arbitrary
13182 }
13183
13184 @[inline] def TermElabM.run (x : TermElabM α) (ctx : Context := mkSomeContext) (s : State := {}) : MetaM (α × State) :=
13185   withConfig setElabConfig (x ctx |>.run s)
13186
13187 @[inline] def TermElabM.run' (x : TermElabM α) (ctx : Context := mkSomeContext) (s : State := {}) : MetaM α :=
13188   (·.1) <$> x.run ctx s
13189
13190 @[inline] def TermElabM.toIO (x : TermElabM α)
13191   (ctxCore : Core.Context) (sCore : Core.State)
13192   (ctxMeta : Meta.Context) (sMeta : Meta.State)
13193   (ctx : Context) (s : State) : IO (α × Core.State × Meta.State × State) := do
13194   let ((a, s), sCore, sMeta) ← (x.run ctx s).toIO ctxCore sCore ctxMeta sMeta
13195   pure (a, sCore, sMeta, s)
13196
13197 instance [MetaEval α] : MetaEval (TermElabM α) where
13198   eval env opts x _ :=
13199     let x : TermElabM α := do
13200       try x finally
13201         let s ← get
13202         s.messages.forM fun msg => do IO.println (← msg.toString)
13203     MetaEval.eval env opts (hideUnit := true) $ x.run' mkSomeContext
13204
13205 unsafe def evalExpr (α) (typeName : Name) (value : Expr) : TermElabM α :=
13206   withoutModifyingEnv do

```

```

13207   let name ← mkFreshUserName ` _tmp
13208   let type ← inferType value
13209   let type ← whnfD type
13210   unless type.isConstOf typeName do
13211     throwError "unexpected type at evalExpr{indentExpr type}"
13212   let decl := Declaration.defnDecl {
13213     name := name, levelParams := [], type := type,
13214     value := value, hints := ReducibilityHints.opaque,
13215     safety := DefinitionSafety.unsafe
13216   }
13217   ensureNoUnassignedMVars decl
13218   addAndCompile decl
13219   evalConst α name
13220
13221 private def throwStuckAtUniverseCnstr : TermElabM Unit := do
13222   let entries ← getPostponed
13223   let mut found : Std.HashSet (Level × Level) := {}
13224   let mut uniqueEntries := #[]
13225   for entry in entries do
13226     let mut lhs := entry.lhs
13227     let mut rhs := entry.rhs
13228     if Level.normLt rhs lhs then
13229       (lhs, rhs) := (rhs, lhs)
13230     unless found.contains (lhs, rhs) do
13231       found := found.insert (lhs, rhs)
13232       uniqueEntries := uniqueEntries.push entry
13233   for i in [1:uniqueEntries.size] do
13234     logErrorAt uniqueEntries[i].ref (← mkMessage uniqueEntries[i])
13235   throwErrorAt uniqueEntries[0].ref (← mkMessage uniqueEntries[0])
13236 where
13237   /- Annotate any constant and sort in `e` that satisfies `p` with `pp.universes true` -/
13238   exposeRelevantUniverses (e : Expr) (p : Level → Bool) : Expr :=
13239     e.replace fun e =>
13240       match e with
13241       | Expr.const _ us _ => if us.any p then some (e.setPPUniverses true) else none
13242       | Expr.sort u _     => if p u then some (e.setPPUniverses true) else none
13243       | _                 => none
13244
13245   mkMessage (entry : PostponedEntry) : TermElabM MessageData := do
13246     match entry.ctx? with
13247     | none =>
13248       return m!"stuck at solving universe constraints{indentD m!"{entry.lhs} =?= {entry.rhs}"}"
13249     | some ctx =>
13250       withLCtx ctx.lctx ctx.localInstances do
13251         let s := entry.lhs.collectMVars entry.rhs.collectMVars
13252         /- `p u` is true if it contains a universe metavariable in `s` -/
13253         let p (u : Level) := u.any fun | Level.mvar m _ => s.contains m | _ => false

```

```

13254     let lhs := exposeRelevantUniverses (← instantiateMVars ctx.lhs) p
13255     let rhs := exposeRelevantUniverses (← instantiateMVars ctx.rhs) p
13256     addMessageContext m!"stuck at solving universe constraints{indentD m!"{entry.lhs} =?= {entry.rhs}}"\nwhile trying to unify{inde
13257
13258 @[specialize] def withoutPostponingUniverseConstraints (x : TermElabM  $\alpha$ ) : TermElabM  $\alpha$  := do
13259     let postponed ← getResetPostponed
13260     try
13261         let a ← x
13262         unless (← processPostponed (mayPostpone := false)) do
13263             throwStuckAtUniverseCnstr
13264             setPostponed postponed
13265         return a
13266     catch ex =>
13267         setPostponed postponed
13268         throw ex
13269
13270 end Term
13271
13272 builtin_initialize
13273     registerTraceClass `Elab.postpone
13274     registerTraceClass `Elab.coe
13275     registerTraceClass `Elab.debug
13276
13277 export Term (TermElabM)
13278
13279 end Lean.Elab
13280 ::::::::::::::
13281 Elab/Util.lean
13282 ::::::::::::::
13283 /-
13284 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
13285 Released under Apache 2.0 license as described in the file LICENSE.
13286 Authors: Leonardo de Moura
13287 -/
13288 import Lean.Util.Trace
13289 import Lean.Parser.Syntax
13290 import Lean.Parser.Extension
13291 import Lean.KeyedDeclsAttribute
13292 import Lean.Elab.Exception
13293
13294 namespace Lean
13295
13296 def Syntax.prettyPrint (stx : Syntax) : Format :=
13297     match stx.unsetTrailing.reprint with -- TODO use syntax pretty printer
13298     | some str => format str.toFormat
13299     | none     => format stx
13300

```

```

13301 def MacroScopesView.format (view : MacroScopesView) (mainModule : Name) : Format :=
13302   fmt $
13303   if view.scopes.isEmpty then
13304     view.name
13305   else if view.mainModule == mainModule then
13306     view.scopes.foldl Name.mkNum (view.name ++ view.imported)
13307   else
13308     view.scopes.foldl Name.mkNum (view.name ++ view.imported ++ view.mainModule)
13309
13310 namespace Elab
13311
13312 def expandOptNamedPrio (stx : Syntax) : MacroM Nat :=
13313   if stx.isNone then
13314     return eval_prio default
13315   else match stx[0] with
13316   | `(Parser.Command.namedPrio| (priority := $prio)) => evalPrio prio
13317   | _ => Macro.throwUnsupported
13318
13319 def expandOptNamedName (stx : Syntax) : MacroM (Option Name) := do
13320   if stx.isNone then
13321     return none
13322   else match stx[0] with
13323   | `(Parser.Command.namedName| (name := $name)) => return name.getId
13324   | _ => Macro.throwUnsupported
13325
13326 structure MacroStackElem where
13327   before : Syntax
13328   after : Syntax
13329
13330 abbrev MacroStack := List MacroStackElem
13331
13332 /- If `ref` does not have position information, then try to use macroStack -/
13333 def getBetterRef (ref : Syntax) (macroStack : MacroStack) : Syntax :=
13334   match ref.getPos? with
13335   | some _ => ref
13336   | none   =>
13337     match macroStack.find? (·.before.getPos? != none) with
13338     | some elem => elem.before
13339     | none      => ref
13340
13341 register_builtin_option pp.macroStack : Bool := {
13342   defValue := false
13343   group   := "pp"
13344   descr   := "display macro expansion stack"
13345 }
13346
13347 def addMacroStack {m} [Monad m] [MonadOptions m] (msgData : MessageData) (macroStack : MacroStack) : m MessageData := do

```

```

13348   if !pp.macroStack.get (← getOptions) then pure msgData else
13349   match macroStack with
13350   | [] => pure msgData
13351   | stack@(top::_) =>
13352     let msgData := msgData ++ Format.line ++ "with resulting expansion" ++ indentD top.after
13353     pure $ stack.foldl
13354       (fun (msgData : MessageData) (elem : MacroStackElem) =>
13355         msgData ++ Format.line ++ "while expanding" ++ indentD elem.before)
13356     msgData
13357
13358 def checkSyntaxNodeKind (k : Name) : AttrM Name := do
13359   if Parser.isValidSyntaxNodeKind (← getEnv) k then pure k
13360   else throwError "failed"
13361
13362 def checkSyntaxNodeKindAtNamespacesAux (k : Name) : Name → AttrM Name
13363   | n@(Name.str p _) => checkSyntaxNodeKind (n ++ k) <|> checkSyntaxNodeKindAtNamespacesAux k p
13364   | _ => throwError "failed"
13365
13366 def checkSyntaxNodeKindAtNamespaces (k : Name) : AttrM Name := do
13367   let ctx ← read
13368   checkSyntaxNodeKindAtNamespacesAux k ctx.currNamespace
13369
13370 def syntaxNodeKindOfAttrParam (defaultParserNamespace : Name) (stx : Syntax) : AttrM SyntaxNodeKind := do
13371   let k ← Attribute.Builtin.getId stx
13372   checkSyntaxNodeKind k
13373   <|>
13374   checkSyntaxNodeKindAtNamespaces k
13375   <|>
13376   checkSyntaxNodeKind (defaultParserNamespace ++ k)
13377   <|>
13378   throwError "invalid syntax node kind '{k}'"
13379
13380 private unsafe def evalSyntaxConstantUnsafe (env : Environment) (opts : Options) (constName : Name) : ExceptT String Id Syntax :=
13381   env.evalConstCheck Syntax opts `Lean.Syntax constName
13382
13383 @[implementedBy evalSyntaxConstantUnsafe]
13384 constant evalSyntaxConstant (env : Environment) (opts : Options) (constName : Name) : ExceptT String Id Syntax := throw ""
13385
13386 unsafe def mkElabAttribute (γ) (attrDeclName attrBuiltinName attrName : Name) (parserNamespace : Name) (typeName : Name) (kind : String
13387   : IO (KeyedDeclsAttribute γ) :=
13388   KeyedDeclsAttribute.init {
13389     builtinName := attrBuiltinName
13390     name := attrName
13391     descr := kind ++ " elaborator"
13392     valueTypeName := typeName
13393     evalKey := fun _ stx => syntaxNodeKindOfAttrParam parserNamespace stx
13394   } attrDeclName

```

```

13395
13396 unsafe def mkMacroAttributeUnsafe : IO (KeyedDeclsAttribute Macro) :=
13397   mkElabAttribute Macro `Lean.Elab.macroAttribute `builtinMacro `macro Name.anonymous `Lean.Macro "macro"
13398
13399 @[implementedBy mkMacroAttributeUnsafe]
13400 constant mkMacroAttribute : IO (KeyedDeclsAttribute Macro)
13401
13402 builtin_initialize macroAttribute : KeyedDeclsAttribute Macro ← mkMacroAttribute
13403
13404 private def expandMacroFns (stx : Syntax) : List Macro → MacroM Syntax
13405 | []      => throw Macro.Exception.unsupportedSyntax
13406 | m::ms => do
13407   try
13408     m stx
13409   catch
13410     | Macro.Exception.unsupportedSyntax => expandMacroFns stx ms
13411     | ex                               => throw ex
13412
13413 def getMacros (env : Environment) : Macro := fun stx =>
13414   let k := stx.getKind
13415   let table := (macroAttribute.ext.getState env).table
13416   match table.find? k with
13417   | some macroFns => expandMacroFns stx macroFns
13418   | none          => throw Macro.Exception.unsupportedSyntax
13419
13420 class MonadMacroAdapter (m : Type → Type) where
13421   getCurrMacroScope      : m MacroScope
13422   getNextMacroScope      : m MacroScope
13423   setNextMacroScope      : MacroScope → m Unit
13424
13425 instance (m n) [MonadLift m n] [MonadMacroAdapter m] : MonadMacroAdapter n := {
13426   getCurrMacroScope := liftM (MonadMacroAdapter.getCurrMacroScope : m _),
13427   getNextMacroScope := liftM (MonadMacroAdapter.getNextMacroScope : m _),
13428   setNextMacroScope := fun s => liftM (MonadMacroAdapter.setNextMacroScope s : m _)
13429 }
13430
13431 private def expandMacro? (env : Environment) (stx : Syntax) : MacroM (Option Syntax) := do
13432   try
13433     let newStx ← getMacros env stx
13434     pure (some newStx)
13435   catch
13436     | Macro.Exception.unsupportedSyntax => pure none
13437     | ex                               => throw ex
13438
13439 @[inline] def liftMacroM {α} {m : Type → Type} [Monad m] [MonadMacroAdapter m] [MonadEnv m] [MonadRecDepth m] [MonadError m] (x : Macro)
13440   let env ← getEnv
13441   match x { macroEnv      := Macro.mkMacroEnv (expandMacro? env),

```

```

13442         ref                := ← getRef,
13443         currMacroScope      := ← MonadMacroAdapter.getCurrMacroScope,
13444         mainModule          := env.mainModule,
13445         currRecDepth        := ← MonadRecDepth.getRecDepth,
13446         maxRecDepth         := ← MonadRecDepth.getMaxRecDepth } (← MonadMacroAdapter.getNextMacroScope) with
13447   | EStateM.Result.error Macro.Exception.unsupportedSyntax _ => throwUnsupportedSyntax
13448   | EStateM.Result.error (Macro.Exception.error ref msg) _   => throwErrorAt ref msg
13449   | EStateM.Result.ok a nextMacroScope                       => MonadMacroAdapter.setNextMacroScope nextMacroScope; pure a
13450
13451 @[inline] def adaptMacro {m : Type → Type} [Monad m] [MonadMacroAdapter m] [MonadEnv m] [MonadRecDepth m] [MonadError m] (x : Macro) (s :
13452   liftMacroM (x stx)
13453
13454 partial def mkUnusedBaseName [Monad m] [MonadEnv m] [MonadResolveName m] (baseName : Name) : m Name := do
13455   let currNamespace ← getCurrNamespace
13456   let env ← getEnv
13457   if env.contains (currNamespace ++ baseName) then
13458     let rec loop (idx : Nat) :=
13459       let name := baseName.appendIndexAfter idx
13460       if env.contains (currNamespace ++ name) then
13461         loop (idx+1)
13462     else
13463       name
13464   return loop 1
13465 else
13466   return baseName
13467
13468 builtin_initialize
13469   registerTraceClass `Elab
13470   registerTraceClass `Elab.step
13471
13472 end Lean.Elab
13473 :::::::::::::::
13474 Parser.lean
13475 :::::::::::::::
13476 /-
13477 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
13478 Released under Apache 2.0 license as described in the file LICENSE.
13479 Authors: Leonardo de Moura, Sebastian Ullrich
13480 -/
13481 import Lean.Parser.Basic
13482 import Lean.Parser.Level
13483 import Lean.Parser.Term
13484 import Lean.Parser.Tactic
13485 import Lean.Parser.Command
13486 import Lean.Parser.Module
13487 import Lean.Parser.Syntax
13488 import Lean.Parser.Do

```

```

13489
13490 namespace Lean
13491 namespace Parser
13492
13493 builtin_initialize
13494   registerAlias "ws" checkWsBefore
13495   registerAlias "noWs" checkNoWsBefore
13496   registerAlias "linebreak" checkLinebreakBefore
13497   registerAlias "num" numLit
13498   registerAlias "str" strLit
13499   registerAlias "char" charLit
13500   registerAlias "name" nameLit
13501   registerAlias "ident" ident
13502   registerAlias "colGt" checkColGt
13503   registerAlias "colGe" checkColGe
13504   registerAlias "lookahead" lookahead
13505   registerAlias "atomic" atomic
13506   registerAlias "many" many
13507   registerAlias "many1" many1
13508   registerAlias "notFollowedBy" (notFollowedBy · "element")
13509   registerAlias "optional" optional
13510   registerAlias "withPosition" withPosition
13511   registerAlias "interpolatedStr" interpolatedStr
13512   registerAlias "orelse" orelse
13513   registerAlias "andthen" andthen
13514
13515 end Parser
13516
13517 namespace PrettyPrinter
13518 namespace Parenthesizer
13519
13520 -- Close the mutual recursion loop; see corresponding `[extern]` in the parenthesizer.
13521 @[export lean_mk_antiquot_parenthesizer]
13522 def mkAntiquot.parenthesizer (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Parenthesizer :=
13523   Parser.mkAntiquot.parenthesizer name kind anonymous
13524
13525 -- The parenthesizer auto-generated these instances correctly, but tagged them with the wrong kind, since the actual kind
13526 -- (e.g. `ident`) is not equal to the parser name `Lean.Parser.Term.ident`.
13527 @[builtinParenthesizer ident] def ident.parenthesizer : Parenthesizer := Parser.Term.ident.parenthesizer
13528 @[builtinParenthesizer numLit] def numLit.parenthesizer : Parenthesizer := Parser.Term.num.parenthesizer
13529 @[builtinParenthesizer scientificLit] def scientificLit.parenthesizer : Parenthesizer := Parser.Term.scientific.parenthesizer
13530 @[builtinParenthesizer charLit] def charLit.parenthesizer : Parenthesizer := Parser.Term.char.parenthesizer
13531 @[builtinParenthesizer strLit] def strLit.parenthesizer : Parenthesizer := Parser.Term.str.parenthesizer
13532
13533 open Lean.Parser
13534
13535 @[export lean_pretty_printer_parenthesizer_interpret_parser_descr]

```



```

13536 unsafe def interpretParserDescr : ParserDescr → CoreM Parenthesizer
13537   | ParserDescr.const n                => getConstAlias parenthesizerAliasesRef n
13538   | ParserDescr.unary n d              => return (← getUnaryAlias parenthesizerAliasesRef n) (← interpretParserDescr d)
13539   | ParserDescr.binary n d₁ d₂        => return (← getBinaryAlias parenthesizerAliasesRef n) (← interpretParserDescr d₁)
13540   | ParserDescr.node k prec d          => return leadingNode.parenthesizer k prec (← interpretParserDescr d)
13541   | ParserDescr.nodeWithAntiquot _ k d => return node.parenthesizer k (← interpretParserDescr d)
13542   | ParserDescr.sepBy p sep psep trail => return sepBy.parenthesizer (← interpretParserDescr p) sep (← interpretParserDescr psep) trail
13543   | ParserDescr.sepBy1 p sep psep trail => return sepBy1.parenthesizer (← interpretParserDescr p) sep (← interpretParserDescr psep) trail
13544   | ParserDescr.trailingNode k prec lhsPrec d => return trailingNode.parenthesizer k prec lhsPrec (← interpretParserDescr d)
13545   | ParserDescr.symbol tk              => return symbol.parenthesizer tk
13546   | ParserDescr.nonReservedSymbol tk includeIdent => return nonReservedSymbol.parenthesizer tk includeIdent
13547   | ParserDescr.parser constName       => combinatorParenthesizerAttribute.runDeclFor constName
13548   | ParserDescr.cat catName prec       => return categoryParser.parenthesizer catName prec
13549
13550 end Parenthesizer
13551
13552 namespace Formatter
13553
13554 @[export lean_mk_antiquot_formatter]
13555 def mkAntiquot.formatter (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Formatter :=
13556   Parser.mkAntiquot.formatter name kind anonymous
13557
13558 @[builtinFormatter ident] def ident.formatter : Formatter := Parser.Term.ident.formatter
13559 @[builtinFormatter numLit] def numLit.formatter : Formatter := Parser.Term.num.formatter
13560 @[builtinFormatter scientificLit] def scientificLit.formatter : Formatter := Parser.Term.scientific.formatter
13561 @[builtinFormatter charLit] def charLit.formatter : Formatter := Parser.Term.char.formatter
13562 @[builtinFormatter strLit] def strLit.formatter : Formatter := Parser.Term.str.formatter
13563
13564 open Lean.Parser
13565
13566 @[export lean_pretty_printer_formatter_interpret_parser_descr]
13567 unsafe def interpretParserDescr : ParserDescr → CoreM Formatter
13568   | ParserDescr.const n                => getConstAlias formatterAliasesRef n
13569   | ParserDescr.unary n d              => return (← getUnaryAlias formatterAliasesRef n) (← interpretParserDescr d)
13570   | ParserDescr.binary n d₁ d₂        => return (← getBinaryAlias formatterAliasesRef n) (← interpretParserDescr d₁) (← interpretParserDescr d₂)
13571   | ParserDescr.node k prec d          => return node.formatter k (← interpretParserDescr d)
13572   | ParserDescr.nodeWithAntiquot _ k d => return node.formatter k (← interpretParserDescr d)
13573   | ParserDescr.sepBy p sep psep trail => return sepBy.formatter (← interpretParserDescr p) sep (← interpretParserDescr psep) trail
13574   | ParserDescr.sepBy1 p sep psep trail => return sepBy1.formatter (← interpretParserDescr p) sep (← interpretParserDescr psep) trail
13575   | ParserDescr.trailingNode k prec lhsPrec d => return trailingNode.formatter k prec lhsPrec (← interpretParserDescr d)
13576   | ParserDescr.symbol tk              => return symbol.formatter tk
13577   | ParserDescr.nonReservedSymbol tk includeIdent => return nonReservedSymbol.formatter tk includeIdent
13578   | ParserDescr.parser constName       => combinatorFormatterAttribute.runDeclFor constName
13579   | ParserDescr.cat catName prec       => return categoryParser.formatter catName prec
13580
13581 end Formatter
13582 end PrettyPrinter

```

```

13583 end Lean
13584 ::::::::::::::
13585 Parser/Attr.lean
13586 ::::::::::::::
13587 /-
13588 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
13589 Released under Apache 2.0 license as described in the file LICENSE.
13590 Authors: Leonardo de Moura
13591 -/
13592 import Lean.Parser.Basic
13593 import Lean.Parser.Extra
13594
13595 namespace Lean.Parser
13596
13597 builtin_initialize
13598   registerBuiltinParserAttribute `builtinPrioParser `prio LeadingIdentBehavior.both
13599   registerBuiltinDynamicParserAttribute `prioParser `prio
13600
13601 builtin_initialize
13602   registerBuiltinParserAttribute `builtinAttrParser `attr LeadingIdentBehavior.symbol
13603   registerBuiltinDynamicParserAttribute `attrParser `attr
13604
13605 @[inline] def priorityParser (rbp : Nat := 0) : Parser :=
13606   categoryParser `prio rbp
13607
13608 @[inline] def attrParser (rbp : Nat := 0) : Parser :=
13609   categoryParser `attr rbp
13610
13611 attribute [runBuiltinParserAttributeHooks]
13612   priorityParser attrParser
13613
13614 namespace Priority
13615 @[builtinPrioParser] def numPrio := checkPrec maxPrec >> numLit
13616 attribute [runBuiltinParserAttributeHooks] numPrio
13617 end Priority
13618
13619 namespace Attr
13620
13621 @[builtinAttrParser] def simple      := leading_parser ident >> optional (priorityParser <|> ident)
13622 /- Remark, We can't use `simple` for `class`, `instance`, `export`, and `macro` because they are keywords. -/
13623 @[builtinAttrParser] def «macro»    := leading_parser "macro " >> ident
13624 @[builtinAttrParser] def «export»   := leading_parser "export " >> ident
13625
13626 /- We don't use `simple` for recursor because the argument is not a priority-/
13627 @[builtinAttrParser] def recursor    := leading_parser nonReservedSymbol "recursor " >> numLit
13628 @[builtinAttrParser] def «class»     := leading_parser "class"
13629 @[builtinAttrParser] def «instance»  := leading_parser "instance" >> optional priorityParser

```

```

13630 @[builtinAttrParser] def defaultInstance := leading_parser nonReservedSymbol "defaultInstance " >> optional priorityParser
13631
13632 def externEntry := leading_parser optional ident >> optional (nonReservedSymbol "inline ") >> strLit
13633 @[builtinAttrParser] def extern      := leading_parser nonReservedSymbol "extern " >> optional numLit >> many externEntry
13634
13635 end Attr
13636
13637 end Lean.Parser
13638 ::::::::::::::
13639 Parser/Basic.lean
13640 ::::::::::::::
13641 /-
13642 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
13643 Released under Apache 2.0 license as described in the file LICENSE.
13644 Authors: Leonardo de Moura, Sebastian Ullrich
13645 -/
13646
13647 /-!
13648 # Basic Lean parser infrastructure
13649
13650 The Lean parser was developed with the following primary goals in mind:
13651
13652 * flexibility: Lean's grammar is complex and includes indentation and other whitespace sensitivity. It should be
13653   possible to introduce such custom "tweaks" locally without having to adjust the fundamental parsing approach.
13654 * extensibility: Lean's grammar can be extended on the fly within a Lean file, and with Lean 4 we want to extend this
13655   to cover embedding domain-specific languages that may look nothing like Lean, down to using a separate set of tokens.
13656 * losslessness: The parser should produce a concrete syntax tree that preserves all whitespace and other "sub-token"
13657   information for the use in tooling.
13658 * performance: The overhead of the parser building blocks, and the overall parser performance on average-complexity
13659   input, should be comparable with that of the previous parser hand-written in C++. No fancy optimizations should be
13660   necessary for this.
13661
13662 Given these constraints, we decided to implement a combinatoric, non-monadic, lexer-less, memoizing recursive-descent
13663 parser. Using combinators instead of some more formal and introspectible grammar representation ensures ultimate
13664 flexibility as well as efficient extensibility: there is (almost) no pre-processing necessary when extending the grammar
13665 with a new parser. However, because the all results the combinators produce are of the homogeneous `Syntax` type, the
13666 basic parser type is not actually a monad but a monomorphic linear function `ParserState → ParserState`, avoiding
13667 constructing and deconstructing countless monadic return values. Instead of explicitly returning syntax objects, parsers
13668 push (zero or more of) them onto a syntax stack inside the linear state. Chaining parsers via `>>` accumulates their
13669 output on the stack. Combinators such as `node` then pop off all syntax objects produced during their invocation and
13670 wrap them in a single `Syntax.node` object that is again pushed on this stack. Instead of calling `node` directly, we
13671 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
13672 declaration being defined.
13673
13674 The lack of a dedicated lexer ensures we can modify and replace the lexical grammar at any point, and simplifies
13675 detecting and propagating whitespace. The parser still has a concept of "tokens", however, and caches the most recent
13676 one for performance: when `tokenFn` is called twice at the same position in the input, it will reuse the result of the

```

```

13677 first call. `tokenFn` recognizes some built-in variable-length tokens such as identifiers as well as any fixed token in
13678 the `ParserContext`'s `TokenTable` (a trie); however, the same cache field and strategy could be reused by custom token
13679 parsers. Tokens also play a central role in the `prattParser` combinator, which selects a *leading* parser followed by
13680 zero or more *trailing* parsers based on the current token (via `peekToken`); see the documentation of `prattParser`
13681 for more details. Tokens are specified via the `symbol` parser, or with `symbolNoWs` for tokens that should not be preceded by whitespa
13682
13683 The `Parser` type is extended with additional metadata over the mere parsing function to propagate token information:
13684 `collectTokens` collects all tokens within a parser for registering. `firstTokens` holds information about the "FIRST"
13685 token set used to speed up parser selection in `prattParser`. This approach of combining static and dynamic information
13686 in the parser type is inspired by the paper "Deterministic, Error-Correcting Combinator Parsers" by Swierstra and Duponcheel.
13687 If multiple parsers accept the same current token, `prattParser` tries all of them using the backtracking `longestMatchFn` combinator.
13688 This is the only case where standard parsers might execute arbitrary backtracking. At the moment there is no memoization shared by thes
13689 parallel parsers apart from the first token, though we might change this in the future if the need arises.
13690
13691 Finally, error reporting follows the standard combinatoric approach of collecting a single unexpected token/... and zero
13692 or more expected tokens (see `Error` below). Expected tokens are e.g. set by `symbol` and merged by `<|>`. Combinators
13693 running multiple parsers should check if an error message is set in the parser state (`hasError`) and act accordingly.
13694 Error recovery is left to the designer of the specific language; for example, Lean's top-level `parseCommand` loop skips
13695 tokens until the next command keyword on error.
13696 -/
13697 import Lean.Data.Trie
13698 import Lean.Data.Position
13699 import Lean.Syntax
13700 import Lean.ToExpr
13701 import Lean.Environment
13702 import Lean.Attributes
13703 import Lean.Message
13704 import Lean.Compiler.InitAttr
13705 import Lean.ResolveName
13706
13707 namespace Lean
13708
13709 namespace Parser
13710
13711 def isLitKind (k : SyntaxNodeKind) : Bool :=
13712   k == strLitKind || k == numLitKind || k == charLitKind || k == nameLitKind || k == scientificLitKind
13713
13714 abbrev mkAtom (info : SourceInfo) (val : String) : Syntax :=
13715   Syntax.atom info val
13716
13717 abbrev mkIdent (info : SourceInfo) (rawVal : Substring) (val : Name) : Syntax :=
13718   Syntax.ident info rawVal val []
13719
13720 /- Return character after position `pos` -/
13721 def getNext (input : String) (pos : Nat) : Char :=
13722   input.get (input.next pos)
13723

```

```

13724 /- Maximal (and function application) precedence.
13725   In the standard lean language, no parser has precedence higher than `maxPrec`.
13726
13727   Note that nothing prevents users from using a higher precedence, but we strongly
13728   discourage them from doing it. -/
13729 def maxPrec   : Nat := eval_prec max
13730 def argPrec   : Nat := eval_prec arg
13731 def leadPrec  : Nat := eval_prec lead
13732 def minPrec   : Nat := eval_prec min
13733
13734 abbrev Token := String
13735
13736 structure TokenCacheEntry where
13737   startPos : String.Pos := 0
13738   stopPos  : String.Pos := 0
13739   token    : Syntax := Syntax.missing
13740
13741 structure ParserCache where
13742   tokenCache : TokenCacheEntry
13743
13744 def initCacheForInput (input : String) : ParserCache := {
13745   tokenCache := { startPos := input.bsize + 1 /- make sure it is not a valid position -/}
13746 }
13747
13748 abbrev TokenTable := Trie Token
13749
13750 abbrev SyntaxNodeKindSet := Std.PersistentHashMap SyntaxNodeKind Unit
13751
13752 def SyntaxNodeKindSet.insert (s : SyntaxNodeKindSet) (k : SyntaxNodeKind) : SyntaxNodeKindSet :=
13753   Std.PersistentHashMap.insert s k ()
13754
13755 /-
13756   Input string and related data. Recall that the `FileMap` is a helper structure for mapping
13757   `String.Pos` in the input string to line/column information. -/
13758 structure InputContext where
13759   input      : String
13760   fileName   : String
13761   fileMap    : FileMap
13762   deriving Inhabited
13763
13764 /- Input context derived from elaboration of previous commands. -/
13765 structure ParserModuleContext where
13766   env        : Environment
13767   options    : Options
13768   -- for name lookup
13769   currNamespace : Name := Name.anonymous
13770   openDecls   : List OpenDecl := []

```

```

13771
13772 structure ParserContext extends InputContext, ParserModuleContext where
13773   prec          : Nat
13774   tokens        : TokenTable
13775   insideQuot    : Bool := false
13776   suppressInsideQuot : Bool := false
13777   savedPos?     : Option String.Pos := none
13778   forbiddenTk?  : Option Token := none
13779
13780 def ParserContext.resolveName (ctx : ParserContext) (id : Name) : List (Name × List String) :=
13781   ResolveName.resolveGlobalName ctx.env ctx.currNamespace ctx.openDecls id
13782
13783 structure Error where
13784   unexpected : String := ""
13785   expected   : List String := []
13786   deriving Inhabited, BEq
13787
13788 namespace Error
13789
13790 private def expectedToString : List String → String
13791 | []      => ""
13792 | [e]     => e
13793 | [e1, e2] => e1 ++ " or " ++ e2
13794 | e::es   => e ++ ", " ++ expectedToString es
13795
13796 protected def toString (e : Error) : String :=
13797   let unexpected := if e.unexpected == "" then [] else [e.unexpected]
13798   let expected   := if e.expected == [] then [] else
13799     let expected := e.expected.toArray.qsort (fun e e' => e < e')
13800     let expected := expected.toList.eraseReps
13801     ["expected " ++ expectedToString expected]
13802   "; ".intercalate $ unexpected ++ expected
13803
13804 instance : ToString Error := ⟨Error.toString⟩
13805
13806 def merge (e1 e2 : Error) : Error :=
13807   match e2 with
13808   | { unexpected := u, .. } => { unexpected := if u == "" then e1.unexpected else u, expected := e1.expected ++ e2.expected }
13809
13810 end Error
13811
13812 structure ParserState where
13813   stxStack : Array Syntax := #[]
13814   /--
13815     Set to the precedence of the preceding (not surrounding) parser by `runLongestMatchParser`
13816     for the use of `checkLhsPrec` in trailing parsers.
13817     Note that with chaining, the preceding parser can be another trailing parser:

```

```

13818     in `1 * 2 + 3`, the preceding parser is '*' when '+' is executed. -/
13819     lhsPrec  : Nat := 0
13820     pos      : String.Pos := 0
13821     cache    : ParserCache
13822     errorMsg : Option Error := none
13823
13824 namespace ParserState
13825
13826 @[inline] def hasError (s : ParserState) : Bool :=
13827   s.errorMsg != none
13828
13829 @[inline] def stackSize (s : ParserState) : Nat :=
13830   s.stxStack.size
13831
13832 def restore (s : ParserState) (iniStackSz : Nat) (iniPos : Nat) : ParserState :=
13833   { s with stxStack := s.stxStack.shrink iniStackSz, errorMsg := none, pos := iniPos }
13834
13835 def setPos (s : ParserState) (pos : Nat) : ParserState :=
13836   { s with pos := pos }
13837
13838 def setCache (s : ParserState) (cache : ParserCache) : ParserState :=
13839   { s with cache := cache }
13840
13841 def pushSyntax (s : ParserState) (n : Syntax) : ParserState :=
13842   { s with stxStack := s.stxStack.push n }
13843
13844 def popSyntax (s : ParserState) : ParserState :=
13845   { s with stxStack := s.stxStack.pop }
13846
13847 def shrinkStack (s : ParserState) (iniStackSz : Nat) : ParserState :=
13848   { s with stxStack := s.stxStack.shrink iniStackSz }
13849
13850 def next (s : ParserState) (input : String) (pos : Nat) : ParserState :=
13851   { s with pos := input.next pos }
13852
13853 def toErrorMsg (ctx : ParserContext) (s : ParserState) : String :=
13854   match s.errorMsg with
13855   | none      => ""
13856   | some msg =>
13857     let pos := ctx.fileMap.toPosition s.pos
13858     mkErrorStringWithPos ctx.fileName pos (toString msg)
13859
13860 def mkNode (s : ParserState) (k : SyntaxNodeKind) (iniStackSz : Nat) : ParserState :=
13861   match s with
13862   | {stack, lhsPrec, pos, cache, err} =>
13863     if err != none && stack.size == iniStackSz then
13864       -- If there is an error but there are no new nodes on the stack, use `missing` instead.

```

```

13865 -- Thus we ensure the property that an syntax tree contains (at least) one `missing` node
13866 -- if (and only if) there was a parse error.
13867 -- We should not create an actual node of kind `k` in this case because it would mean we
13868 -- choose an "arbitrary" node (in practice the last one) in an alternative of the form
13869 -- `node k1 p1 <|> ... <|> node kn pn` when all parsers fail. With the code below we
13870 -- instead return a less misleading single `missing` node without randomly selecting any `ki`.
13871 let stack := stack.push Syntax.missing
13872 (stack, lhsPrec, pos, cache, err)
13873 else
13874   let newNode := Syntax.node k (stack.extract iniStackSz stack.size)
13875   let stack := stack.shrink iniStackSz
13876   let stack := stack.push newNode
13877   (stack, lhsPrec, pos, cache, err)
13878
13879 def mkTrailingNode (s : ParserState) (k : SyntaxNodeKind) (iniStackSz : Nat) : ParserState :=
13880   match s with
13881   | (stack, lhsPrec, pos, cache, err) =>
13882     let newNode := Syntax.node k (stack.extract (iniStackSz - 1) stack.size)
13883     let stack := stack.shrink (iniStackSz - 1)
13884     let stack := stack.push newNode
13885     (stack, lhsPrec, pos, cache, err)
13886
13887 def mkError (s : ParserState) (msg : String) : ParserState :=
13888   match s with
13889   | (stack, lhsPrec, pos, cache, _) => (stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ] })
13890
13891 def mkUnexpectedError (s : ParserState) (msg : String) (expected : List String := []) : ParserState :=
13892   match s with
13893   | (stack, lhsPrec, pos, cache, _) => (stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg, expected := expected })
13894
13895 def mkEOIError (s : ParserState) (expected : List String := []) : ParserState :=
13896   s.mkUnexpectedError "unexpected end of input" expected
13897
13898 def mkErrorAt (s : ParserState) (msg : String) (pos : String.Pos) (initStackSz? : Option Nat := none) : ParserState :=
13899   match s, initStackSz? with
13900   | (stack, lhsPrec, _, cache, _), none => (stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ] })
13901   | (stack, lhsPrec, _, cache, _), some sz => (stack.shrink sz |>.push Syntax.missing, lhsPrec, pos, cache, some { expected := [ msg ] })
13902
13903 def mkErrorsAt (s : ParserState) (ex : List String) (pos : String.Pos) (initStackSz? : Option Nat := none) : ParserState :=
13904   match s, initStackSz? with
13905   | (stack, lhsPrec, _, cache, _), none => (stack.push Syntax.missing, lhsPrec, pos, cache, some { expected := ex })
13906   | (stack, lhsPrec, _, cache, _), some sz => (stack.shrink sz |>.push Syntax.missing, lhsPrec, pos, cache, some { expected := ex })
13907
13908 def mkUnexpectedErrorAt (s : ParserState) (msg : String) (pos : String.Pos) : ParserState :=
13909   match s with
13910   | (stack, lhsPrec, _, cache, _) => (stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg })
13911

```



```

13912 end ParserState
13913
13914 def ParserFn := ParserContext → ParserState → ParserState
13915
13916 instance : Inhabited ParserFn where
13917   default := fun ctx s => s
13918
13919 inductive FirstTokens where
13920   | epsilon    : FirstTokens
13921   | unknown    : FirstTokens
13922   | tokens     : List Token → FirstTokens
13923   | optTokens  : List Token → FirstTokens
13924   deriving Inhabited
13925
13926 namespace FirstTokens
13927
13928 def seq : FirstTokens → FirstTokens → FirstTokens
13929   | epsilon,      tks      => tks
13930   | optTokens s1, optTokens s2 => optTokens (s1 ++ s2)
13931   | optTokens s1, tokens s2   => tokens (s1 ++ s2)
13932   | tks,          _        => tks
13933
13934 def toOptional : FirstTokens → FirstTokens
13935   | tokens tks => optTokens tks
13936   | tks        => tks
13937
13938 def merge : FirstTokens → FirstTokens → FirstTokens
13939   | epsilon,      tks      => toOptional tks
13940   | tks,          epsilon  => toOptional tks
13941   | tokens s1,   tokens s2 => tokens (s1 ++ s2)
13942   | optTokens s1, optTokens s2 => optTokens (s1 ++ s2)
13943   | tokens s1,   optTokens s2 => optTokens (s1 ++ s2)
13944   | optTokens s1, tokens s2   => optTokens (s1 ++ s2)
13945   | _,           _        => unknown
13946
13947 def toStr : FirstTokens → String
13948   | epsilon    => "epsilon"
13949   | unknown    => "unknown"
13950   | tokens tks  => toString tks
13951   | optTokens tks => "?" ++ toString tks
13952
13953 instance : ToString FirstTokens := {toStr}
13954
13955 end FirstTokens
13956
13957 structure ParserInfo where
13958   collectTokens : List Token → List Token := id

```

```

13959 collectKinds : SyntaxNodeKindSet → SyntaxNodeKindSet := id
13960 firstTokens   : FirstTokens := FirstTokens.unknown
13961 deriving Inhabited
13962
13963 structure Parser where
13964   info : ParserInfo := {}
13965   fn   : ParserFn
13966   deriving Inhabited
13967
13968 abbrev TrailingParser := Parser
13969
13970 def dbgTraceStateFn (label : String) (p : ParserFn) : ParserFn :=
13971   fun c s =>
13972     let sz := s.stxStack.size
13973     let s' := p c s
13974     dbg_trace "{label}
13975 pos: {s'.pos}
13976 err: {s'.errorMsg}
13977 out: {s'.stxStack.extract sz s'.stxStack.size}" s'
13978
13979 def dbgTraceState (label : String) (p : Parser) : Parser where
13980   fn := dbgTraceStateFn label p.fn
13981   info := p.info
13982
13983 @[noinline] def epsilonInfo : ParserInfo :=
13984   { firstTokens := FirstTokens.epsilon }
13985
13986 @[inline] def checkStackTopFn (p : Syntax → Bool) (msg : String) : ParserFn := fun c s =>
13987   if p s.stxStack.back then s
13988   else s.mkUnexpectedError msg
13989
13990 @[inline] def checkStackTop (p : Syntax → Bool) (msg : String) : Parser := {
13991   info := epsilonInfo,
13992   fn   := checkStackTopFn p msg
13993 }
13994
13995 @[inline] def andthenFn (p q : ParserFn) : ParserFn := fun c s =>
13996   let s := p c s
13997   if s.hasError then s else q c s
13998
13999 @[noinline] def andthenInfo (p q : ParserInfo) : ParserInfo := {
14000   collectTokens := p.collectTokens ◦ q.collectTokens,
14001   collectKinds  := p.collectKinds ◦ q.collectKinds,
14002   firstTokens   := p.firstTokens.seq q.firstTokens
14003 }
14004
14005 @[inline] def andthen (p q : Parser) : Parser := {

```

```

14006   info := andthenInfo p.info q.info,
14007   fn    := andthenFn p.fn q.fn
14008 }
14009
14010 instance : AndThen Parser := (andthen)
14011
14012 @[inline] def nodeFn (n : SyntaxNodeKind) (p : ParserFn) : ParserFn := fun c s =>
14013   let iniSz := s.stackSize
14014   let s      := p c s
14015   s.mkNode n iniSz
14016
14017 @[inline] def trailingNodeFn (n : SyntaxNodeKind) (p : ParserFn) : ParserFn := fun c s =>
14018   let iniSz := s.stackSize
14019   let s      := p c s
14020   s.mkTrailingNode n iniSz
14021
14022 @[noinline] def nodeInfo (n : SyntaxNodeKind) (p : ParserInfo) : ParserInfo := {
14023   collectTokens := p.collectTokens,
14024   collectKinds  := fun s => (p.collectKinds s).insert n,
14025   firstTokens   := p.firstTokens
14026 }
14027
14028 @[inline] def node (n : SyntaxNodeKind) (p : Parser) : Parser := {
14029   info := nodeInfo n p.info,
14030   fn   := nodeFn n p.fn
14031 }
14032
14033 def errorFn (msg : String) : ParserFn := fun _ s =>
14034   s.mkUnexpectedError msg
14035
14036 @[inline] def error (msg : String) : Parser := {
14037   info := epsilonInfo,
14038   fn   := errorFn msg
14039 }
14040
14041 def errorAtSavedPosFn (msg : String) (delta : Bool) : ParserFn := fun c s =>
14042   match c.savedPos? with
14043   | none    => s
14044   | some pos =>
14045     let pos := if delta then c.input.next pos else pos
14046     match s with
14047     | (stack, lhsPrec, _, cache, _) => (stack.push Syntax.missing, lhsPrec, pos, cache, some { unexpected := msg })
14048
14049 /- Generate an error at the position saved with the `withPosition` combinator.
14050    If `delta == true`, then it reports at saved position+1.
14051    This useful to make sure a parser consumed at least one character. -/
14052 @[inline] def errorAtSavedPos (msg : String) (delta : Bool) : Parser := {

```

```

14053   fn := errorAtSavedPosFn msg delta
14054 }
14055
14056 /- Succeeds if `c.prec <= prec` -/
14057 def checkPrecFn (prec : Nat) : ParserFn := fun c s =>
14058   if c.prec <= prec then s
14059   else s.mkUnexpectedError "unexpected token at this precedence level; consider parenthesizing the term"
14060
14061 @[inline] def checkPrec (prec : Nat) : Parser := {
14062   info := epsilonInfo,
14063   fn   := checkPrecFn prec
14064 }
14065
14066 /- Succeeds if `c.lhsPrec >= prec` -/
14067 def checkLhsPrecFn (prec : Nat) : ParserFn := fun c s =>
14068   if s.lhsPrec >= prec then s
14069   else s.mkUnexpectedError "unexpected token at this precedence level; consider parenthesizing the term"
14070
14071 @[inline] def checkLhsPrec (prec : Nat) : Parser := {
14072   info := epsilonInfo,
14073   fn   := checkLhsPrecFn prec
14074 }
14075
14076 def setLhsPrecFn (prec : Nat) : ParserFn := fun c s =>
14077   if s.hasError then s
14078   else { s with lhsPrec := prec }
14079
14080 @[inline] def setLhsPrec (prec : Nat) : Parser := {
14081   info := epsilonInfo,
14082   fn   := setLhsPrecFn prec
14083 }
14084
14085 def checkInsideQuotFn : ParserFn := fun c s =>
14086   if c.insideQuot then s
14087   else s.mkUnexpectedError "unexpected syntax outside syntax quotation"
14088
14089 @[inline] def checkInsideQuot : Parser := {
14090   info := epsilonInfo,
14091   fn   := checkInsideQuotFn
14092 }
14093
14094 def checkOutsideQuotFn : ParserFn := fun c s =>
14095   if !c.insideQuot then s
14096   else s.mkUnexpectedError "unexpected syntax inside syntax quotation"
14097
14098 @[inline] def checkOutsideQuot : Parser := {
14099   info := epsilonInfo,

```

```

14100 fn := checkOutsideQuotFn
14101 }
14102
14103 def toggleInsideQuotFn (p : ParserFn) : ParserFn := fun c s =>
14104   if c.suppressInsideQuot then p c s
14105   else p { c with insideQuot := !c.insideQuot } s
14106
14107 @[inline] def toggleInsideQuot (p : Parser) : Parser := {
14108   info := p.info,
14109   fn := toggleInsideQuotFn p.fn
14110 }
14111
14112 def suppressInsideQuotFn (p : ParserFn) : ParserFn := fun c s =>
14113   p { c with suppressInsideQuot := true } s
14114
14115 @[inline] def suppressInsideQuot (p : Parser) : Parser := {
14116   info := p.info,
14117   fn := suppressInsideQuotFn p.fn
14118 }
14119
14120 @[inline] def leadingNode (n : SyntaxNodeKind) (prec : Nat) (p : Parser) : Parser :=
14121   checkPrec prec >> node n p >> setLhsPrec prec
14122
14123 @[inline] def trailingNodeAux (n : SyntaxNodeKind) (p : Parser) : TrailingParser := {
14124   info := nodeInfo n p.info,
14125   fn := trailingNodeFn n p.fn
14126 }
14127
14128 @[inline] def trailingNode (n : SyntaxNodeKind) (prec lhsPrec : Nat) (p : Parser) : TrailingParser :=
14129   checkPrec prec >> checkLhsPrec lhsPrec >> trailingNodeAux n p >> setLhsPrec prec
14130
14131 def mergeOrElseErrors (s : ParserState) (error1 : Error) (iniPos : Nat) (mergeErrors : Bool) : ParserState :=
14132   match s with
14133   | (stack, lhsPrec, pos, cache, some error2) =>
14134     if pos == iniPos then (stack, lhsPrec, pos, cache, some (if mergeErrors then error1.merge error2 else error2))
14135     else s
14136   | other => other
14137
14138 def orelseFnCore (p q : ParserFn) (mergeErrors : Bool) : ParserFn := fun c s =>
14139   let iniSz := s.stackSize
14140   let iniPos := s.pos
14141   let s := p c s
14142   match s.errorMsg with
14143   | some errorMsg =>
14144     if s.pos == iniPos then
14145       mergeOrElseErrors (q c (s.restore iniSz iniPos)) errorMsg iniPos mergeErrors
14146   else

```

```

14147     s
14148   | none => s
14149
14150 @[inline] def orelseFn (p q : ParserFn) : ParserFn :=
14151   orelseFnCore p q true
14152
14153 @[noinline] def orelseInfo (p q : ParserInfo) : ParserInfo := {
14154   collectTokens := p.collectTokens ◦ q.collectTokens,
14155   collectKinds  := p.collectKinds ◦ q.collectKinds,
14156   firstTokens   := p.firstTokens.merge q.firstTokens
14157 }
14158
14159 /--
14160   Run `p`, falling back to `q` if `p` failed without consuming any input.
14161
14162   NOTE: In order for the pretty printer to retrace an `orelse`, `p` must be a call to `node` or some other parser
14163   producing a single node kind. Nested `orelse` calls are flattened for this, i.e. `(node k1 p1 <|> node k2 p2) <|> ...`
14164   is fine as well. -/
14165 @[inline] def orelse (p q : Parser) : Parser := {
14166   info := orelseInfo p.info q.info,
14167   fn   := orelseFn p.fn q.fn
14168 }
14169
14170 instance : OrElse Parser := {orelse}
14171
14172 @[noinline] def noFirstTokenInfo (info : ParserInfo) : ParserInfo := {
14173   collectTokens := info.collectTokens,
14174   collectKinds  := info.collectKinds
14175 }
14176
14177 def atomicFn (p : ParserFn) : ParserFn := fun c s =>
14178   let iniPos := s.pos
14179   match p c s with
14180   | (stack, lhsPrec, _, cache, some msg) => (stack, lhsPrec, iniPos, cache, some msg)
14181   | other                               => other
14182
14183 @[inline] def atomic (p : Parser) : Parser := {
14184   info := p.info,
14185   fn   := atomicFn p.fn
14186 }
14187
14188 def optionalFn (p : ParserFn) : ParserFn := fun c s =>
14189   let iniSz  := s.stackSize
14190   let iniPos := s.pos
14191   let s      := p c s
14192   let s      := if s.hasError && s.pos == iniPos then s.restore iniSz iniPos else s
14193   s.mkNode nullKind iniSz

```

```

14194
14195 @[noinline] def optionaInfo (p : ParserInfo) : ParserInfo := {
14196   collectTokens := p.collectTokens,
14197   collectKinds  := p.collectKinds,
14198   firstTokens   := p.firstTokens.toOptional
14199 }
14200
14201 @[inline] def optionalNoAntiquot (p : Parser) : Parser := {
14202   info := optionaInfo p.info,
14203   fn    := optionalFn p.fn
14204 }
14205
14206 def lookaheadFn (p : ParserFn) : ParserFn := fun c s =>
14207   let iniSz := s.stackSize
14208   let iniPos := s.pos
14209   let s      := p c s
14210   if s.hasError then s else s.restore iniSz iniPos
14211
14212 @[inline] def lookahead (p : Parser) : Parser := {
14213   info := p.info,
14214   fn    := lookaheadFn p.fn
14215 }
14216
14217 def notFollowedByFn (p : ParserFn) (msg : String) : ParserFn := fun c s =>
14218   let iniSz := s.stackSize
14219   let iniPos := s.pos
14220   let s      := p c s
14221   if s.hasError then
14222     s.restore iniSz iniPos
14223   else
14224     let s := s.restore iniSz iniPos
14225     s.mkUnexpectedError s!"unexpected {msg}"
14226
14227 @[inline] def notFollowedBy (p : Parser) (msg : String) : Parser := {
14228   fn := notFollowedByFn p.fn msg
14229 }
14230
14231 partial def manyAux (p : ParserFn) : ParserFn := fun c s => do
14232   let iniSz := s.stackSize
14233   let iniPos := s.pos
14234   let mut s  := p c s
14235   if s.hasError then
14236     return if iniPos == s.pos then s.restore iniSz iniPos else s
14237   if iniPos == s.pos then
14238     return s.mkUnexpectedError "invalid 'many' parser combinator application, parser did not consume anything"
14239   if s.stackSize > iniSz + 1 then
14240     s := s.mkNode nullKind iniSz

```

```

14241 manyAux p c s
14242
14243 @[inline] def manyFn (p : ParserFn) : ParserFn := fun c s =>
14244   let iniSz := s.stackSize
14245   let s := manyAux p c s
14246   s.mkNode nullKind iniSz
14247
14248 @[inline] def manyNoAntiquot (p : Parser) : Parser := {
14249   info := noFirstTokenInfo p.info,
14250   fn := manyFn p.fn
14251 }
14252
14253 @[inline] def many1Fn (p : ParserFn) : ParserFn := fun c s =>
14254   let iniSz := s.stackSize
14255   let s := andthenFn p (manyAux p) c s
14256   s.mkNode nullKind iniSz
14257
14258 @[inline] def many1NoAntiquot (p : Parser) : Parser := {
14259   info := p.info,
14260   fn := many1Fn p.fn
14261 }
14262
14263 private partial def sepByFnAux (p : ParserFn) (sep : ParserFn) (allowTrailingSep : Bool) (iniSz : Nat) (pOpt : Bool) : ParserFn :=
14264   let rec parse (pOpt : Bool) (c s) := do
14265     let sz := s.stackSize
14266     let pos := s.pos
14267     let mut s := p c s
14268     if s.hasError then
14269       if s.pos > pos then
14270         return s.mkNode nullKind iniSz
14271       else if pOpt then
14272         let s := s.restore sz pos
14273         return s.mkNode nullKind iniSz
14274     else
14275       -- append `Syntax.missing` to make clear that List is incomplete
14276       let s := s.pushSyntax Syntax.missing
14277       return s.mkNode nullKind iniSz
14278   if s.stackSize > sz + 1 then
14279     s := s.mkNode nullKind sz
14280   let sz := s.stackSize
14281   let pos := s.pos
14282   let s := sep c s
14283   if s.hasError then
14284     let s := s.restore sz pos
14285     return s.mkNode nullKind iniSz
14286   if s.stackSize > sz + 1 then
14287     s := s.mkNode nullKind sz

```



```

14288     parse allowTrailingSep c s
14289     parse pOpt
14290
14291 def sepByFn (allowTrailingSep : Bool) (p : ParserFn) (sep : ParserFn) : ParserFn := fun c s =>
14292     let iniSz := s.stackSize
14293     sepByFnAux p sep allowTrailingSep iniSz true c s
14294
14295 def sepBy1Fn (allowTrailingSep : Bool) (p : ParserFn) (sep : ParserFn) : ParserFn := fun c s =>
14296     let iniSz := s.stackSize
14297     sepByFnAux p sep allowTrailingSep iniSz false c s
14298
14299 @[noinline] def sepByInfo (p sep : ParserInfo) : ParserInfo := {
14300     collectTokens := p.collectTokens ◦ sep.collectTokens,
14301     collectKinds  := p.collectKinds ◦ sep.collectKinds
14302 }
14303
14304 @[noinline] def sepBy1Info (p sep : ParserInfo) : ParserInfo := {
14305     collectTokens := p.collectTokens ◦ sep.collectTokens,
14306     collectKinds  := p.collectKinds ◦ sep.collectKinds,
14307     firstTokens   := p.firstTokens
14308 }
14309
14310 @[inline] def sepByNoAntiquot (p sep : Parser) (allowTrailingSep : Bool := false) : Parser := {
14311     info := sepByInfo p.info sep.info,
14312     fn   := sepByFn allowTrailingSep p.fn sep.fn
14313 }
14314
14315 @[inline] def sepBy1NoAntiquot (p sep : Parser) (allowTrailingSep : Bool := false) : Parser := {
14316     info := sepBy1Info p.info sep.info,
14317     fn   := sepBy1Fn allowTrailingSep p.fn sep.fn
14318 }
14319
14320 /- Apply `f` to the syntax object produced by `p` -/
14321 def withResultOfFn (p : ParserFn) (f : Syntax → Syntax) : ParserFn := fun c s =>
14322     let s := p c s
14323     if s.hasError then s
14324     else
14325         let stx := s.stxStack.back
14326         s.popSyntax.pushSyntax (f stx)
14327
14328 @[noinline] def withResultOfInfo (p : ParserInfo) : ParserInfo := {
14329     collectTokens := p.collectTokens,
14330     collectKinds  := p.collectKinds
14331 }
14332
14333 @[inline] def withResultOf (p : Parser) (f : Syntax → Syntax) : Parser := {
14334     info := withResultOfInfo p.info,

```

```

14335   fn    := withResultOfFn p.fn f
14336 }
14337
14338 @[inline] def many1Unbox (p : Parser) : Parser :=
14339   withResultOf (many1NoAntiquot p) fun stx => if stx.getNumArgs == 1 then stx.getArg 0 else stx
14340
14341 partial def satisfyFn (p : Char → Bool) (errorMsg : String := "unexpected character") : ParserFn := fun c s =>
14342   let i := s.pos
14343   if c.input.atEnd i then s.mkEOIError
14344   else if p (c.input.get i) then s.next c.input i
14345   else s.mkUnexpectedError errorMsg
14346
14347 partial def takeUntilFn (p : Char → Bool) : ParserFn := fun c s =>
14348   let i := s.pos
14349   if c.input.atEnd i then s
14350   else if p (c.input.get i) then s
14351   else takeUntilFn p c (s.next c.input i)
14352
14353 def takeWhileFn (p : Char → Bool) : ParserFn :=
14354   takeUntilFn (fun c => !p c)
14355
14356 @[inline] def takeWhile1Fn (p : Char → Bool) (errorMsg : String) : ParserFn :=
14357   andthenFn (satisfyFn p errorMsg) (takeWhileFn p)
14358
14359 variable (startPos : String.Pos) in
14360 partial def finishCommentBlock (nesting : Nat) : ParserFn := fun c s =>
14361   let input := c.input
14362   let i     := s.pos
14363   if input.atEnd i then eoi s
14364   else
14365     let curr := input.get i
14366     let i     := input.next i
14367     if curr == '-' then
14368       if input.atEnd i then eoi s
14369       else
14370         let curr := input.get i
14371         if curr == '/' then -- "-/" end of comment
14372           if nesting == 1 then s.next input i
14373           else finishCommentBlock (nesting-1) c (s.next input i)
14374         else
14375           finishCommentBlock nesting c (s.next input i)
14376     else if curr == '/' then
14377       if input.atEnd i then eoi s
14378       else
14379         let curr := input.get i
14380         if curr == '-' then finishCommentBlock (nesting+1) c (s.next input i)
14381         else finishCommentBlock nesting c (s.setPos i)

```

```

14382     else finishCommentBlock nesting c (s.setPos i)
14383 where
14384   eoi s := s.mkUnexpectedErrorAt "unterminated comment" startPos
14385
14386   /- Consume whitespace and comments -/
14387   partial def whitespace : ParserFn := fun c s =>
14388     let input := c.input
14389     let i     := s.pos
14390     if input.atEnd i then s
14391   else
14392     let curr := input.get i
14393     if curr.isWhitespace then whitespace c (s.next input i)
14394     else if curr == '-' then
14395       let i := input.next i
14396       let curr := input.get i
14397       if curr == '-' then andthenFn (takeUntilFn (fun c => c == '\n')) whitespace c (s.next input i)
14398     else s
14399     else if curr == '/' then
14400       let startPos := i
14401       let i := input.next i
14402       let curr := input.get i
14403       if curr == '-' then
14404         let i := input.next i
14405         let curr := input.get i
14406         if curr == '-' then s -- /--" doc comment is an actual token
14407         else andthenFn (finishCommentBlock startPos 1) whitespace c (s.next input i)
14408       else s
14409     else s
14410
14411   def mkEmptySubstringAt (s : String) (p : Nat) : Substring :=
14412     { str := s, startPos := p, stopPos := p }
14413
14414   private def rawAux (startPos : Nat) (trailingWs : Bool) : ParserFn := fun c s =>
14415     let input := c.input
14416     let stopPos := s.pos
14417     let leading := mkEmptySubstringAt input startPos
14418     let val := input.extract startPos stopPos
14419     if trailingWs then
14420       let s := whitespace c s
14421       let stopPos' := s.pos
14422       let trailing := { str := input, startPos := stopPos, stopPos := stopPos' : Substring }
14423       let atom := mkAtom (SourceInfo.original leading startPos trailing) val
14424       s.pushSyntax atom
14425     else
14426       let trailing := mkEmptySubstringAt input stopPos
14427       let atom := mkAtom (SourceInfo.original leading startPos trailing) val
14428       s.pushSyntax atom

```

```

14429
14430 /-- Match an arbitrary Parser and return the consumed String in a `Syntax.atom`. -/
14431 @[inline] def rawFn (p : ParserFn) (trailingWs := false) : ParserFn := fun c s =>
14432   let startPos := s.pos
14433   let s := p c s
14434   if s.hasError then s else rawAux startPos trailingWs c s
14435
14436 @[inline] def chFn (c : Char) (trailingWs := false) : ParserFn :=
14437   rawFn (satisfyFn (fun d => c == d) ("'" ++ toString c ++ "'")) trailingWs
14438
14439 def rawCh (c : Char) (trailingWs := false) : Parser :=
14440   { fn := chFn c trailingWs }
14441
14442 def hexDigitFn : ParserFn := fun c s =>
14443   let input := c.input
14444   let i := s.pos
14445   if input.atEnd i then s.mkEOLError
14446   else
14447     let curr := input.get i
14448     let i := input.next i
14449     if curr.isDigit || ('a' <= curr && curr <= 'f') || ('A' <= curr && curr <= 'F') then s.setPos i
14450     else s.mkUnexpectedError "invalid hexadecimal numeral"
14451
14452 def quotedCharCoreFn (isQuotable : Char → Bool) : ParserFn := fun c s =>
14453   let input := c.input
14454   let i := s.pos
14455   if input.atEnd i then s.mkEOLError
14456   else
14457     let curr := input.get i
14458     if isQuotable curr then
14459       s.next input i
14460     else if curr == 'x' then
14461       andthenFn hexDigitFn hexDigitFn c (s.next input i)
14462     else if curr == 'u' then
14463       andthenFn hexDigitFn (andthenFn hexDigitFn (andthenFn hexDigitFn hexDigitFn)) c (s.next input i)
14464     else
14465       s.mkUnexpectedError "invalid escape sequence"
14466
14467 def isQuotableCharDefault (c : Char) : Bool :=
14468   c == '\\' || c == '\"' || c == '\'' || c == 'r' || c == 'n' || c == 't'
14469
14470 def quotedCharFn : ParserFn :=
14471   quotedCharCoreFn isQuotableCharDefault
14472
14473 /-- Push `(Syntax.node tk <new-atom>)` into syntax stack -/
14474 def mkNodeToken (n : SyntaxNodeKind) (startPos : Nat) : ParserFn := fun c s =>
14475   let input := c.input

```

```

14476 let stopPos := s.pos
14477 let leading := mkEmptySubstringAt input startPos
14478 let val := input.extract startPos stopPos
14479 let s := whitespace c s
14480 let wsStopPos := s.pos
14481 let trailing := { str := input, startPos := stopPos, stopPos := wsStopPos : Substring }
14482 let info := SourceInfo.original leading startPos trailing
14483 s.pushSyntax (Syntax.mkLit n val info)
14484
14485 def charLitFnAux (startPos : Nat) : ParserFn := fun c s =>
14486   let input := c.input
14487   let i := s.pos
14488   if input.atEnd i then s.mkEOLError
14489   else
14490     let curr := input.get i
14491     let s := s.setPos (input.next i)
14492     let s := if curr == '\\' then quotedCharFn c s else s
14493     if s.hasError then s
14494     else
14495       let i := s.pos
14496       let curr := input.get i
14497       let s := s.setPos (input.next i)
14498       if curr == '\\' then mkNodeToken charLitKind startPos c s
14499       else s.mkUnexpectedError "missing end of character literal"
14500
14501 partial def strLitFnAux (startPos : Nat) : ParserFn := fun c s =>
14502   let input := c.input
14503   let i := s.pos
14504   if input.atEnd i then s.mkUnexpectedErrorAt "unterminated string literal" startPos
14505   else
14506     let curr := input.get i
14507     let s := s.setPos (input.next i)
14508     if curr == '\"' then
14509       mkNodeToken strLitKind startPos c s
14510     else if curr == '\\' then andthenFn quotedCharFn (strLitFnAux startPos) c s
14511     else strLitFnAux startPos c s
14512
14513 def decimalNumberFn (startPos : Nat) (c : ParserContext) : ParserState → ParserState := fun s =>
14514   let s := takeWhileFn (fun c => c.isDigit) c s
14515   let input := c.input
14516   let i := s.pos
14517   let curr := input.get i
14518   if curr == '.' || curr == 'e' || curr == 'E' then
14519     let s := parseOptDot s
14520     let s := parseOptExp s
14521     mkNodeToken scientificLitKind startPos c s
14522   else

```

```

14523     mkNodeToken numLitKind startPos c s
14524 where
14525     parseOptDot s :=
14526         let input := c.input
14527         let i     := s.pos
14528         let curr  := input.get i
14529         if curr == '.' then
14530             let i := input.next i
14531             let curr := input.get i
14532             if curr.isDigit then
14533                 takeWhileFn (fun c => c.isDigit) c (s.setPos i)
14534             else
14535                 s.setPos i
14536         else
14537             s
14538
14539     parseOptExp s :=
14540         let input := c.input
14541         let i     := s.pos
14542         let curr  := input.get i
14543         if curr == 'e' || curr == 'E' then
14544             let i := input.next i
14545             let i := if input.get i == '-' then input.next i else i
14546             let curr := input.get i
14547             if curr.isDigit then
14548                 takeWhileFn (fun c => c.isDigit) c (s.setPos i)
14549             else
14550                 s.setPos i
14551         else
14552             s
14553
14554 def binNumberFn (startPos : Nat) : ParserFn := fun c s =>
14555     let s := takeWhile1Fn (fun c => c == '0' || c == '1') "binary number" c s
14556     mkNodeToken numLitKind startPos c s
14557
14558 def octalNumberFn (startPos : Nat) : ParserFn := fun c s =>
14559     let s := takeWhile1Fn (fun c => '0' ≤ c && c ≤ '7') "octal number" c s
14560     mkNodeToken numLitKind startPos c s
14561
14562 def hexNumberFn (startPos : Nat) : ParserFn := fun c s =>
14563     let s := takeWhile1Fn (fun c => ('0' ≤ c && c ≤ '9') || ('a' ≤ c && c ≤ 'f') || ('A' ≤ c && c ≤ 'F')) "hexadecimal number" c s
14564     mkNodeToken numLitKind startPos c s
14565
14566 def numberFnAux : ParserFn := fun c s =>
14567     let input := c.input
14568     let startPos := s.pos
14569     if input.atEnd startPos then s.mkEOIError

```

```

14570 else
14571   let curr := input.get startPos
14572   if curr == '0' then
14573     let i := input.next startPos
14574     let curr := input.get i
14575     if curr == 'b' || curr == 'B' then
14576       binNumberFn startPos c (s.next input i)
14577     else if curr == 'o' || curr == 'O' then
14578       octalNumberFn startPos c (s.next input i)
14579     else if curr == 'x' || curr == 'X' then
14580       hexNumberFn startPos c (s.next input i)
14581     else
14582       decimalNumberFn startPos c (s.setPos i)
14583   else if curr.isDigit then
14584     decimalNumberFn startPos c (s.next input startPos)
14585   else
14586     s.mkError "numeral"
14587
14588 def isIdCont : String → ParserState → Bool := fun input s =>
14589   let i := s.pos
14590   let curr := input.get i
14591   if curr == '.' then
14592     let i := input.next i
14593     if input.atEnd i then
14594       false
14595     else
14596       let curr := input.get i
14597       isIdFirst curr || isIdBeginEscape curr
14598   else
14599     false
14600
14601 private def isToken (idStartPos idStopPos : Nat) (tk : Option Token) : Bool :=
14602   match tk with
14603   | none => false
14604   | some tk =>
14605     -- if a token is both a symbol and a valid identifier (i.e. a keyword),
14606     -- we want it to be recognized as a symbol
14607     tk.bsize ≥ idStopPos - idStartPos
14608
14609
14610 def mkTokenAndFixPos (startPos : Nat) (tk : Option Token) : ParserFn := fun c s =>
14611   match tk with
14612   | none => s.mkErrorAt "token" startPos
14613   | some tk =>
14614     if c.forbiddenTk? == some tk then
14615       s.mkErrorAt "forbidden token" startPos
14616   else

```

```

14617     let input      := c.input
14618     let leading    := mkEmptySubstringAt input startPos
14619     let stopPos    := startPos + tk.bsize
14620     let s          := s.setPos stopPos
14621     let s          := whitespace c s
14622     let wsStopPos  := s.pos
14623     let trailing   := { str := input, startPos := stopPos, stopPos := wsStopPos : Substring }
14624     let atom       := mkAtom (SourceInfo.original leading startPos trailing) tk
14625     s.pushSyntax atom
14626
14627 def mkIdResult (startPos : Nat) (tk : Option Token) (val : Name) : ParserFn := fun c s =>
14628   let stopPos      := s.pos
14629   if isToken startPos stopPos tk then
14630     mkTokenAndFixPos startPos tk c s
14631   else
14632     let input      := c.input
14633     let rawVal     := { str := input, startPos := startPos, stopPos := stopPos : Substring }
14634     let s          := whitespace c s
14635     let trailingStopPos := s.pos
14636     let leading    := mkEmptySubstringAt input startPos
14637     let trailing   := { str := input, startPos := stopPos, stopPos := trailingStopPos : Substring }
14638     let info       := SourceInfo.original leading startPos trailing
14639     let atom       := mkIdent info rawVal val
14640     s.pushSyntax atom
14641
14642 partial def identFnAux (startPos : Nat) (tk : Option Token) (r : Name) : ParserFn :=
14643   let rec parse (r : Name) (c s) := do
14644     let input := c.input
14645     let i     := s.pos
14646     if input.atEnd i then
14647       return s.mkEOIError
14648     let curr := input.get i
14649     if isIdBeginEscape curr then
14650       let startPart := input.next i
14651       let s         := takeUntilFn isIdEndEscape c (s.setPos startPart)
14652       if input.atEnd s.pos then
14653         return s.mkUnexpectedErrorAt "unterminated identifier escape" startPart
14654       let stopPart := s.pos
14655       let s        := s.next c.input s.pos
14656       let r := Name.mkStr r (input.extract startPart stopPart)
14657       if isIdCont input s then
14658         let s := s.next input s.pos
14659         parse r c s
14660       else
14661         mkIdResult startPos tk r c s
14662     else if isIdFirst curr then
14663       let startPart := i

```



```

14664     let s      := takeWhileFn isIdRest c (s.next input i)
14665     let stopPart := s.pos
14666     let r := Name.mkStr r (input.extract startPart stopPart)
14667     if isIdCont input s then
14668         let s := s.next input s.pos
14669         parse r c s
14670     else
14671         mkIdResult startPos tk r c s
14672     else
14673         mkTokenAndFixPos startPos tk c s
14674     parse r
14675
14676 private def isIdFirstOrBeginEscape (c : Char) : Bool :=
14677     isIdFirst c || isIdBeginEscape c
14678
14679 private def nameLitAux (startPos : Nat) : ParserFn := fun c s =>
14680     let input := c.input
14681     let s      := identFnAux startPos none Name.anonymous c (s.next input startPos)
14682     if s.hasError then
14683         s
14684     else
14685         let stx := s.stxStack.back
14686         match stx with
14687         | Syntax.ident _ rawStr _ _ =>
14688             let s := s.popSyntax
14689             s.pushSyntax (Syntax.node nameLitKind #[mkAtomFrom stx rawStr.toString])
14690         | _ => s.mkError "invalid Name literal"
14691
14692 private def tokenFnAux : ParserFn := fun c s =>
14693     let input := c.input
14694     let i      := s.pos
14695     let curr   := input.get i
14696     if curr == '\"' then
14697         strLitFnAux i c (s.next input i)
14698     else if curr == '\'' then
14699         charLitFnAux i c (s.next input i)
14700     else if curr.isDigit then
14701         numberFnAux c s
14702     else if curr == '`' && isIdFirstOrBeginEscape (getNext input i) then
14703         nameLitAux i c s
14704     else
14705         let (_, tk) := c.tokens.matchPrefix input i
14706         identFnAux i tk Name.anonymous c s
14707
14708 private def updateCache (startPos : Nat) (s : ParserState) : ParserState :=
14709     -- do not cache token parsing errors, which are rare and usually fatal and thus not worth an extra field in `TokenCache`
14710     match s with

```

```

14711 | (stack, lhsPrec, pos, cache, none) =>
14712   if stack.size == 0 then s
14713   else
14714     let tk := stack.back
14715     (stack, lhsPrec, pos, { tokenCache := { startPos := startPos, stopPos := pos, token := tk } }, none)
14716 | other => other
14717
14718 def tokenFn (expected : List String := []) : ParserFn := fun c s =>
14719   let input := c.input
14720   let i     := s.pos
14721   if input.atEnd i then s.mkEOLError expected
14722   else
14723     let tkc := s.cache.tokenCache
14724     if tkc.startPos == i then
14725       let s := s.pushSyntax tkc.token
14726       s.setPos tkc.stopPos
14727     else
14728       let s := tokenFnAux c s
14729       updateCache i s
14730
14731 def peekTokenAux (c : ParserContext) (s : ParserState) : ParserState × Except ParserState Syntax :=
14732   let iniSz := s.stackSize
14733   let iniPos := s.pos
14734   let s      := tokenFn [] c s
14735   if let some e := s.errorMsg then (s.restore iniSz iniPos, Except.error s)
14736   else
14737     let stx := s.stxStack.back
14738     (s.restore iniSz iniPos, Except.ok stx)
14739
14740 def peekToken (c : ParserContext) (s : ParserState) : ParserState × Except ParserState Syntax :=
14741   let tkc := s.cache.tokenCache
14742   if tkc.startPos == s.pos then
14743     (s, Except.ok tkc.token)
14744   else
14745     peekTokenAux c s
14746
14747 14747 /- Treat keywords as identifiers. -/
14748 def rawIdentFn : ParserFn := fun c s =>
14749   let input := c.input
14750   let i     := s.pos
14751   if input.atEnd i then s.mkEOLError
14752   else identFnAux i none Name.anonymous c s
14753
14754 @[inline] def satisfySymbolFn (p : String → Bool) (expected : List String) : ParserFn := fun c s =>
14755   let initStackSz := s.stackSize
14756   let startPos := s.pos
14757   let s        := tokenFn expected c s

```

```

14758   if s.hasError then
14759     s
14760   else
14761     match s.stxStack.back with
14762     | Syntax.atom _ sym => if p sym then s else s.mkErrorsAt expected startPos initStackSz
14763     | _                 => s.mkErrorsAt expected startPos initStackSz
14764
14765 def symbolFnAux (sym : String) (errorMsg : String) : ParserFn :=
14766   satisfySymbolFn (fun s => s == sym) [errorMsg]
14767
14768 def symbolInfo (sym : String) : ParserInfo := {
14769   collectTokens := fun tks => sym :: tks,
14770   firstTokens   := FirstTokens.tokens [ sym ]
14771 }
14772
14773 @[inline] def symbolFn (sym : String) : ParserFn :=
14774   symbolFnAux sym ("'" ++ sym ++ "'")
14775
14776 @[inline] def symbolNoAntiquot (sym : String) : Parser :=
14777   let sym := sym.trim
14778   { info := symbolInfo sym,
14779     fn   := symbolFn sym }
14780
14781 def checkTailNoWs (prev : Syntax) : Bool :=
14782   match prev.getTailInfo with
14783   | SourceInfo.original _ _ trailing => trailing.stopPos == trailing.startPos
14784   | _                               => false
14785
14786 /-- Check if the following token is the symbol_or_identifier `sym`. Useful for
14787      parsing local tokens that have not been added to the token table (but may have
14788      been so by some unrelated code).
14789
14790      For example, the universe `max` Function is parsed using this combinator so that
14791      it can still be used as an identifier outside of universes (but registering it
14792      as a token in a Term Syntax would not break the universe Parser). -/
14793 def nonReservedSymbolFnAux (sym : String) (errorMsg : String) : ParserFn := fun c s =>
14794   let initStackSz := s.stackSize
14795   let startPos := s.pos
14796   let s := tokenFn [errorMsg] c s
14797   if s.hasError then s
14798   else
14799     match s.stxStack.back with
14800     | Syntax.atom _ sym' =>
14801       if sym == sym' then s else s.mkErrorAt errorMsg startPos initStackSz
14802     | Syntax.ident info rawVal _ _ =>
14803       if sym == rawVal.toString then
14804         let s := s.popSyntax

```

```

14805         s.pushSyntax (Syntax.atom info sym)
14806     else
14807         s.mkErrorAt errorMsg startPos initStackSz
14808 | _ => s.mkErrorAt errorMsg startPos initStackSz
14809
14810 @[inline] def nonReservedSymbolFn (sym : String) : ParserFn :=
14811     nonReservedSymbolFnAux sym ("'' ++ sym ++ ''")
14812
14813 def nonReservedSymbolInfo (sym : String) (includeIdent : Bool) : ParserInfo := {
14814     firstTokens :=
14815         if includeIdent then
14816             FirstTokens.tokens [ sym, "ident" ]
14817         else
14818             FirstTokens.tokens [ sym ]
14819 }
14820
14821 @[inline] def nonReservedSymbolNoAntiquot (sym : String) (includeIdent := false) : Parser :=
14822     let sym := sym.trim
14823     { info := nonReservedSymbolInfo sym includeIdent,
14824       fn   := nonReservedSymbolFn sym }
14825
14826 partial def strAux (sym : String) (errorMsg : String) (j : Nat) : ParserFn :=
14827     let rec parse (j c s) :=
14828         if sym.atEnd j then s
14829         else
14830             let i := s.pos
14831             let input := c.input
14832             if input.atEnd i || sym.get j != input.get i then s.mkError errorMsg
14833             else parse (sym.next j) c (s.next input i)
14834     parse j
14835
14836 def checkTailWs (prev : Syntax) : Bool :=
14837     match prev.getTailInfo with
14838     | SourceInfo.original _ _ trailing => trailing.stopPos > trailing.startPos
14839     | _                               => false
14840
14841 def checkWsBeforeFn (errorMsg : String) : ParserFn := fun c s =>
14842     let prev := s.stxStack.back
14843     if checkTailWs prev then s else s.mkError errorMsg
14844
14845 def checkWsBefore (errorMsg : String := "space before") : Parser := {
14846     info := epsilonInfo,
14847     fn   := checkWsBeforeFn errorMsg
14848 }
14849
14850 def checkTailLinebreak (prev : Syntax) : Bool :=
14851     match prev.getTailInfo with

```

```

14852 | SourceInfo.original _ _ trailing => trailing.contains '\n'
14853 | _ => false
14854
14855 def checkLinebreakBeforeFn (errorMsg : String) : ParserFn := fun c s =>
14856   let prev := s.stxStack.back
14857   if checkTailLinebreak prev then s else s.mkError errorMsg
14858
14859 def checkLinebreakBefore (errorMsg : String := "line break") : Parser := {
14860   info := epsilonInfo
14861   fn    := checkLinebreakBeforeFn errorMsg
14862 }
14863
14864 private def pickNonNone (stack : Array Syntax) : Syntax :=
14865   match stack.findRev? $ fun stx => !stx.isNone with
14866   | none => Syntax.missing
14867   | some stx => stx
14868
14869 def checkNoWsBeforeFn (errorMsg : String) : ParserFn := fun c s =>
14870   let prev := pickNonNone s.stxStack
14871   if checkTailNoWs prev then s else s.mkError errorMsg
14872
14873 def checkNoWsBefore (errorMsg : String := "no space before") : Parser := {
14874   info := epsilonInfo,
14875   fn    := checkNoWsBeforeFn errorMsg
14876 }
14877
14878 def unicodeSymbolFnAux (sym asciiSym : String) (expected : List String) : ParserFn :=
14879   satisfySymbolFn (fun s => s == sym || s == asciiSym) expected
14880
14881 def unicodeSymbolInfo (sym asciiSym : String) : ParserInfo := {
14882   collectTokens := fun tks => sym :: asciiSym :: tks,
14883   firstTokens   := FirstTokens.tokens [ sym, asciiSym ]
14884 }
14885
14886 @[inline] def unicodeSymbolFn (sym asciiSym : String) : ParserFn :=
14887   unicodeSymbolFnAux sym asciiSym ["'" ++ sym ++ "'", "'" ++ asciiSym ++ "'"]
14888
14889 @[inline] def unicodeSymbolNoAntiquot (sym asciiSym : String) : Parser :=
14890   let sym := sym.trim
14891   let asciiSym := asciiSym.trim
14892   { info := unicodeSymbolInfo sym asciiSym,
14893     fn    := unicodeSymbolFn sym asciiSym }
14894
14895 def mkAtomicInfo (k : String) : ParserInfo :=
14896   { firstTokens := FirstTokens.tokens [ k ] }
14897
14898 def numLitFn : ParserFn :=

```

```

14899 fun c s =>
14900   let initStackSz := s.stackSize
14901   let iniPos := s.pos
14902   let s := tokenFn ["numeral"] c s
14903   if !s.hasError && !(s.stxStack.back.isOfKind numLitKind) then s.mkErrorAt "numeral" iniPos initStackSz else s
14904
14905 @[inline] def numLitNoAntiquot : Parser := {
14906   fn := numLitFn,
14907   info := mkAtomicInfo "numLit"
14908 }
14909
14910 def scientificLitFn : ParserFn :=
14911   fun c s =>
14912     let initStackSz := s.stackSize
14913     let iniPos := s.pos
14914     let s := tokenFn ["scientific number"] c s
14915     if !s.hasError && !(s.stxStack.back.isOfKind scientificLitKind) then s.mkErrorAt "scientific number" iniPos initStackSz else s
14916
14917 @[inline] def scientificLitNoAntiquot : Parser := {
14918   fn := scientificLitFn,
14919   info := mkAtomicInfo "scientificLit"
14920 }
14921
14922 def strLitFn : ParserFn := fun c s =>
14923   let initStackSz := s.stackSize
14924   let iniPos := s.pos
14925   let s := tokenFn ["string literal"] c s
14926   if !s.hasError && !(s.stxStack.back.isOfKind strLitKind) then s.mkErrorAt "string literal" iniPos initStackSz else s
14927
14928 @[inline] def strLitNoAntiquot : Parser := {
14929   fn := strLitFn,
14930   info := mkAtomicInfo "strLit"
14931 }
14932
14933 def charLitFn : ParserFn := fun c s =>
14934   let initStackSz := s.stackSize
14935   let iniPos := s.pos
14936   let s := tokenFn ["char literal"] c s
14937   if !s.hasError && !(s.stxStack.back.isOfKind charLitKind) then s.mkErrorAt "character literal" iniPos initStackSz else s
14938
14939 @[inline] def charLitNoAntiquot : Parser := {
14940   fn := charLitFn,
14941   info := mkAtomicInfo "charLit"
14942 }
14943
14944 def nameLitFn : ParserFn := fun c s =>
14945   let initStackSz := s.stackSize

```

```

14946 let iniPos := s.pos
14947 let s := tokenFn ["Name literal"] c s
14948 if !s.hasError && !(s.stxStack.back.isOfKind nameLitKind) then s.mkErrorAt "Name literal" iniPos initStackSz else s
14949
14950 @[inline] def nameLitNoAntiquot : Parser := {
14951   fn := nameLitFn,
14952   info := mkAtomicInfo "nameLit"
14953 }
14954
14955 def identFn : ParserFn := fun c s =>
14956   let initStackSz := s.stackSize
14957   let iniPos := s.pos
14958   let s := tokenFn ["identifier"] c s
14959   if !s.hasError && !(s.stxStack.back.isIdent) then s.mkErrorAt "identifier" iniPos initStackSz else s
14960
14961 @[inline] def identNoAntiquot : Parser := {
14962   fn := identFn,
14963   info := mkAtomicInfo "ident"
14964 }
14965
14966 @[inline] def rawIdentNoAntiquot : Parser := {
14967   fn := rawIdentFn
14968 }
14969
14970 def identEqFn (id : Name) : ParserFn := fun c s =>
14971   let initStackSz := s.stackSize
14972   let iniPos := s.pos
14973   let s := tokenFn ["identifier"] c s
14974   if s.hasError then
14975     s
14976   else match s.stxStack.back with
14977   | Syntax.ident _ _ val _ => if val != id then s.mkErrorAt ("expected identifier '" ++ toString id ++ "'") iniPos initStackSz else s
14978   | _ => s.mkErrorAt "identifier" iniPos initStackSz
14979
14980 @[inline] def identEq (id : Name) : Parser := {
14981   fn := identEqFn id,
14982   info := mkAtomicInfo "ident"
14983 }
14984
14985 namespace ParserState
14986
14987 def keepTop (s : Array Syntax) (startStackSize : Nat) : Array Syntax :=
14988   let node := s.back
14989   s.shrink startStackSize |>.push node
14990
14991 def keepNewError (s : ParserState) (oldStackSize : Nat) : ParserState :=
14992   match s with

```

```

14993 | (stack, lhsPrec, pos, cache, err) => (keepTop stack oldStackSize, lhsPrec, pos, cache, err)
14994
14995 def keepPrevError (s : ParserState) (oldStackSize : Nat) (oldStopPos : String.Pos) (oldError : Option Error) : ParserState :=
14996   match s with
14997   | (stack, lhsPrec, _, cache, _) => (stack.shrink oldStackSize, lhsPrec, oldStopPos, cache, oldError)
14998
14999 def mergeErrors (s : ParserState) (oldStackSize : Nat) (oldError : Error) : ParserState :=
15000   match s with
15001   | (stack, lhsPrec, pos, cache, some err) =>
15002     if oldError == err then s
15003     else (stack.shrink oldStackSize, lhsPrec, pos, cache, some (oldError.merge err))
15004   | other => other
15005
15006 def keepLatest (s : ParserState) (startStackSize : Nat) : ParserState :=
15007   match s with
15008   | (stack, lhsPrec, pos, cache, _) => (keepTop stack startStackSize, lhsPrec, pos, cache, none)
15009
15010 def replaceLongest (s : ParserState) (startStackSize : Nat) : ParserState :=
15011   s.keepLatest startStackSize
15012
15013 end ParserState
15014
15015 def invalidLongestMatchParser (s : ParserState) : ParserState :=
15016   s.mkError "longestMatch parsers must generate exactly one Syntax node"
15017
15018 /-
15019   Auxiliary function used to execute parsers provided to `longestMatchFn`.
15020   Push `left?` into the stack if it is not `none`, and execute `p`.
15021
15022   Remark: `p` must produce exactly one syntax node.
15023   Remark: the `left?` is not none when we are processing trailing parsers. -/
15024 def runLongestMatchParser (left? : Option Syntax) (startLhsPrec : Nat) (p : ParserFn) : ParserFn := fun c s => do
15025   /-
15026     We assume any registered parser `p` has one of two forms:
15027     * a direct call to `leadingParser` or `trailingParser`
15028     * a direct call to a (leading) token parser
15029     In the first case, we can extract the precedence of the parser by having `leadingParser/trailingParser`
15030     set `ParserState.lhsPrec` to it in the very end so that no nested parser can interfere.
15031     In the second case, the precedence is effectively `max` (there is a `checkPrec` merely for the convenience
15032     of the pretty printer) and there are no nested `leadingParser/trailingParser` calls, so the value of `lhsPrec`
15033     will not be changed by the parser (nor will it be read by any leading parser). Thus we initialize the field
15034     to `maxPrec` in the leading case. -/
15035   let mut s := { s with lhsPrec := if left?.isSome then startLhsPrec else maxPrec }
15036   let startSize := s.stackSize
15037   if let some left := left? then
15038     s := s.pushSyntax left
15039   s := p c s

```



```

15040 -- stack contains `[..., result ]`
15041 if s.stackSize == startSize + 1 then
15042   s -- success or error with the expected number of nodes
15043 else if s.hasError then
15044   -- error with an unexpected number of nodes.
15045   s.shrinkStack startSize |>.pushSyntax Syntax.missing
15046 else
15047   -- parser succeeded with incorrect number of nodes
15048   invalidLongestMatchParser s
15049
15050 def longestMatchStep (left? : Option Syntax) (startSize startLhsPrec : Nat) (startPos : String.Pos) (prevPrio : Nat) (prio : Nat) (p : |
15051   : ParserContext → ParserState → ParserState × Nat := fun c s =>
15052   let prevErrorMsg := s.errorMsg
15053   let prevStopPos := s.pos
15054   let prevSize := s.stackSize
15055   let prevLhsPrec := s.lhsPrec
15056   let s := s.restore prevSize startPos
15057   let s := runLongestMatchParser left? startLhsPrec p c s
15058   match prevErrorMsg, s.errorMsg with
15059   | none, none => -- both succeeded
15060     if s.pos > prevStopPos || (s.pos == prevStopPos && prio > prevPrio) then (s.replaceLongest startSize, prio)
15061     else if s.pos < prevStopPos || (s.pos == prevStopPos && prio < prevPrio) then ({ s.restore prevSize prevStopPos with lhsPrec := pre
15062       -- it is not clear what the precedence of a choice node should be, so we conservatively take the minimum
15063       else ({s with lhsPrec := s.lhsPrec.min prevLhsPrec }, prio)
15064   | none, some _ => -- prev succeeded, current failed
15065     ({ s.restore prevSize prevStopPos with lhsPrec := prevLhsPrec }, prevPrio)
15066   | some oldError, some _ => -- both failed
15067     if s.pos > prevStopPos || (s.pos == prevStopPos && prio > prevPrio) then (s.keepNewError startSize, prio)
15068     else if s.pos < prevStopPos || (s.pos == prevStopPos && prio < prevPrio) then (s.keepPrevError prevSize prevStopPos prevErrorMsg, p
15069     else (s.mergeErrors prevSize oldError, prio)
15070   | some _, none => -- prev failed, current succeeded
15071     let successNode := s.stxStack.back
15072     let s := s.shrinkStack startSize -- restore stack to initial size to make sure (failure) nodes are removed from the stack
15073     (s.pushSyntax successNode, prio) -- put successNode back on the stack
15074
15075 def longestMatchMkResult (startSize : Nat) (s : ParserState) : ParserState :=
15076   if !s.hasError && s.stackSize > startSize + 1 then s.mkNode choiceKind startSize else s
15077
15078 def longestMatchFnAux (left? : Option Syntax) (startSize startLhsPrec : Nat) (startPos : String.Pos) (prevPrio : Nat) (ps : List (Parse
15079   let rec parse (prevPrio : Nat) (ps : List (Parser × Nat)) :=
15080     match ps with
15081     | [] => fun _ s => longestMatchMkResult startSize s
15082     | p::ps => fun c s =>
15083       let (s, prevPrio) := longestMatchStep left? startSize startLhsPrec startPos prevPrio p.2 p.1.fn c s
15084       parse prevPrio ps c s
15085   parse prevPrio ps
15086

```

```

15087 def longestMatchFn (left? : Option Syntax) : List (Parser × Nat) → ParserFn
15088 | [] => fun _ s => s.mkError "longestMatch: empty list"
15089 | [p] => fun c s => runLongestMatchParser left? s.lhsPrec p.1.fn c s
15090 | p::ps => fun c s =>
15091   let startSize := s.stackSize
15092   let startLhsPrec := s.lhsPrec
15093   let startPos := s.pos
15094   let s := runLongestMatchParser left? s.lhsPrec p.1.fn c s
15095   longestMatchFnAux left? startSize startLhsPrec startPos p.2 ps c s
15096
15097 def anyOfFn : List Parser → ParserFn
15098 | [], _, s => s.mkError "anyOf: empty list"
15099 | [p], c, s => p.fn c s
15100 | p::ps, c, s => orelseFn p.fn (anyOfFn ps) c s
15101
15102 @[inline] def checkColGeFn (errorMsg : String) : ParserFn := fun c s =>
15103   match c.savedPos? with
15104   | none => s
15105   | some savedPos =>
15106     let savedPos := c.fileMap.toPosition savedPos
15107     let pos := c.fileMap.toPosition s.pos
15108     if pos.column ≥ savedPos.column then s
15109     else s.mkError errorMsg
15110
15111 @[inline] def checkColGe (errorMsg : String := "checkColGe") : Parser :=
15112   { fn := checkColGeFn errorMsg }
15113
15114 @[inline] def checkColGtFn (errorMsg : String) : ParserFn := fun c s =>
15115   match c.savedPos? with
15116   | none => s
15117   | some savedPos =>
15118     let savedPos := c.fileMap.toPosition savedPos
15119     let pos := c.fileMap.toPosition s.pos
15120     if pos.column > savedPos.column then s
15121     else s.mkError errorMsg
15122
15123 @[inline] def checkColGt (errorMsg : String := "checkColGt") : Parser :=
15124   { fn := checkColGtFn errorMsg }
15125
15126 @[inline] def checkLineEqFn (errorMsg : String) : ParserFn := fun c s =>
15127   match c.savedPos? with
15128   | none => s
15129   | some savedPos =>
15130     let savedPos := c.fileMap.toPosition savedPos
15131     let pos := c.fileMap.toPosition s.pos
15132     if pos.line == savedPos.line then s
15133     else s.mkError errorMsg

```

```

15134
15135 @[inline] def checkLineEq (errorMsg : String := "checkLineEq") : Parser :=
15136   { fn := checkLineEqFn errorMsg }
15137
15138 @[inline] def withPosition (p : Parser) : Parser := {
15139   info := p.info,
15140   fn   := fun c s =>
15141     p.fn { c with savedPos? := s.pos } s
15142 }
15143
15144 @[inline] def withoutPosition (p : Parser) : Parser := {
15145   info := p.info,
15146   fn   := fun c s =>
15147     let pos := c.fileMap.toPosition s.pos
15148     p.fn { c with savedPos? := none } s
15149 }
15150
15151 @[inline] def withForbidden (tk : Token) (p : Parser) : Parser := {
15152   info := p.info,
15153   fn   := fun c s => p.fn { c with forbiddenTk? := tk } s
15154 }
15155
15156 @[inline] def withoutForbidden (p : Parser) : Parser := {
15157   info := p.info,
15158   fn   := fun c s => p.fn { c with forbiddenTk? := none } s
15159 }
15160
15161 def eoiFn : ParserFn := fun c s =>
15162   let i := s.pos
15163   if c.input.atEnd i then s
15164   else s.mkError "expected end of file"
15165
15166 @[inline] def eoi : Parser :=
15167   { fn := eoiFn }
15168
15169 open Std (RMap RMap.empty)
15170
15171 /-- A multimap indexed by tokens. Used for indexing parsers by their leading token. -/
15172 def TokenMap (α : Type) := RMap Name (List α) Name.quickLt
15173
15174 namespace TokenMap
15175
15176 def insert (map : TokenMap α) (k : Name) (v : α) : TokenMap α :=
15177   match map.find? k with
15178   | none   => Std.RMap.insert map k [v]
15179   | some vs => Std.RMap.insert map k (v::vs)
15180

```

```

15181 instance : Inhabited (TokenMap  $\alpha$ ) := (RMap.empty)
15182
15183 instance : EmptyCollection (TokenMap  $\alpha$ ) := (RMap.empty)
15184
15185 end TokenMap
15186
15187 structure PrattParsingTables where
15188   leadingTable      : TokenMap (Parser  $\times$  Nat) := {}
15189   leadingParsers    : List (Parser  $\times$  Nat) := [] -- for supporting parsers we cannot obtain first token
15190   trailingTable     : TokenMap (Parser  $\times$  Nat) := {}
15191   trailingParsers   : List (Parser  $\times$  Nat) := [] -- for supporting parsers such as function application
15192
15193 instance : Inhabited PrattParsingTables := ({} )
15194
15195 /-
15196   The type `leadingIdentBehavior` specifies how the parsing table
15197   lookup function behaves for identifiers. The function `prattParser`
15198   uses two tables `leadingTable` and `trailingTable`. They map tokens
15199   to parsers.
15200
15201   - `LeadingIdentBehavior.default`: if the leading token
15202     is an identifier, then `prattParser` just executes the parsers
15203     associated with the auxiliary token "ident".
15204
15205   - `LeadingIdentBehavior.symbol`: if the leading token is
15206     an identifier ``, and there are parsers `P` associated with
15207     the token ``, then it executes `P`. Otherwise, it executes
15208     only the parsers associated with the auxiliary token "ident".
15209
15210   - `LeadingIdentBehavior.both`: if the leading token
15211     is an identifier ``, then it executes the parsers associated
15212     with token `` and parsers associated with the auxiliary
15213     token "ident".
15214
15215   We use `LeadingIdentBehavior.symbol` and `LeadingIdentBehavior.both`
15216   and `nonReservedSymbol` parser to implement the `tactic` parsers.
15217   The idea is to avoid creating a reserved symbol for each
15218   builtin tactic (e.g., `apply`, `assumption`, etc.). That is, users
15219   may still use these symbols as identifiers (e.g., naming a
15220   function).
15221 -/
15222
15223 inductive LeadingIdentBehavior where
15224   | default
15225   | symbol
15226   | both
15227   deriving Inhabited, BEq

```

```

15228
15229
15230 /--
15231   Each parser category is implemented using a Pratt's parser.
15232   The system comes equipped with the following categories: `level`, `term`, `tactic`, and `command`.
15233   Users and plugins may define extra categories.
15234
15235   The method
15236   ```
15237   categoryParser `term prec
15238   ```
15239   executes the Pratt's parser for category `term` with precedence `prec`.
15240   That is, only parsers with precedence at least `prec` are considered.
15241   The method `termParser prec` is equivalent to the method above.
15242 -/
15243 structure ParserCategory where
15244   tables      : PrattParsingTables
15245   behavior    : LeadingIdentBehavior
15246   deriving    Inhabited
15247
15248 abbrev ParserCategories := Std.PersistentHashMap Name ParserCategory
15249
15250 def indexed {α : Type} (map : TokenMap α) (c : ParserContext) (s : ParserState) (behavior : LeadingIdentBehavior) : ParserState × List α :=
15251   let (s, stx) := peekToken c s
15252   let find (n : Name) : ParserState × List α :=
15253     match map.find? n with
15254     | some as => (s, as)
15255     | _       => (s, [])
15256   match stx with
15257   | Except.ok (Syntax.atom _ sym)      => find (Name.mkSimple sym)
15258   | Except.ok (Syntax.ident _ _ val _) =>
15259     match behavior with
15260     | LeadingIdentBehavior.default => find identKind
15261     | LeadingIdentBehavior.symbol =>
15262       match map.find? val with
15263       | some as => (s, as)
15264       | none    => find identKind
15265     | LeadingIdentBehavior.both =>
15266       match map.find? val with
15267       | some as => match map.find? identKind with
15268         | some as' => (s, as ++ as')
15269         | _       => (s, as)
15270       | none    => find identKind
15271   | Except.ok (Syntax.node k _)      => find k
15272   | Except.ok _                     => (s, [])
15273   | Except.error s'                  => (s', [])
15274

```

```

15275 abbrev CategoryParserFn := Name → ParserFn
15276
15277 builtin_initialize categoryParserFnRef : IO.Ref CategoryParserFn ← IO.mkRef fun _ => whitespace
15278
15279 builtin_initialize categoryParserFnExtension : EnvExtension CategoryParserFn ← registerEnvExtension $ categoryParserFnRef.get
15280
15281 def categoryParserFn (catName : Name) : ParserFn := fun ctx s =>
15282   categoryParserFnExtension.getState ctx.env catName ctx s
15283
15284 def categoryParser (catName : Name) (prec : Nat) : Parser := {
15285   fn := fun c s => categoryParserFn catName { c with prec := prec } s
15286 }
15287
15288 -- Define `termParser` here because we need it for antiquotations
15289 @[inline] def termParser (prec : Nat := 0) : Parser :=
15290   categoryParser `term prec
15291
15292 /- ===== -/
15293 /- Antiquotations -/
15294 /- ===== -/
15295
15296 /-- Fail if previous token is immediately followed by ':'. -/
15297 def checkNoImmediateColon : Parser := {
15298   fn := fun c s =>
15299     let prev := s.stxStack.back
15300     if checkTailNoWs prev then
15301       let input := c.input
15302       let i := s.pos
15303       if input.atEnd i then s
15304     else
15305       let curr := input.get i
15306       if curr == ':' then
15307         s.mkUnexpectedError "unexpected ':'"
15308       else s
15309   else s
15310 }
15311
15312 def setExpectedFn (expected : List String) (p : ParserFn) : ParserFn := fun c s =>
15313   match p c s with
15314   | s'@{ errorMsg := some msg, .. } => { s' with errorMsg := some { msg with expected := [] } }
15315   | s'                               => s'
15316
15317 def setExpected (expected : List String) (p : Parser) : Parser :=
15318   { fn := setExpectedFn expected p.fn, info := p.info }
15319
15320 def pushNone : Parser :=
15321   { fn := fun c s => s.pushSyntax mkNullNode }

```

```

15322
15323 -- We support two kinds of antiquotations: `$$id` and `$(t)`, where `id` is a term identifier and `t` is a term.
15324 def antiquotNestedExpr : Parser := node `antiquotNestedExpr` (symbolNoAntiquot "(" >> toggleInsideQuot termParser >> symbolNoAntiquot ")")
15325 def antiquotExpr : Parser      := identNoAntiquot <|> antiquotNestedExpr
15326
15327 @[inline] def tokenWithAntiquotFn (p : ParserFn) : ParserFn := fun c s => do
15328   let s := p c s
15329   if s.hasError then
15330     return s
15331   let iniSz := s.stackSize
15332   let iniPos := s.pos
15333   let s      := (checkNoWsBefore >> symbolNoAntiquot "%" >> symbolNoAntiquot "$" >> checkNoWsBefore >> antiquotExpr).fn c s
15334   if s.hasError then
15335     return s.restore iniSz iniPos
15336   s.mkNode (`token_antiquot) (iniSz - 1)
15337
15338 @[inline] def tokenWithAntiquot (p : Parser) : Parser where
15339   fn := tokenWithAntiquotFn p.fn
15340   info := p.info
15341
15342 @[inline] def symbol (sym : String) : Parser :=
15343   tokenWithAntiquot (symbolNoAntiquot sym)
15344
15345 instance : Coe String Parser := {fun s => symbol s }
15346
15347 @[inline] def nonReservedSymbol (sym : String) (includeIdent := false) : Parser :=
15348   tokenWithAntiquot (nonReservedSymbolNoAntiquot sym includeIdent)
15349
15350 @[inline] def unicodeSymbol (sym asciiSym : String) : Parser :=
15351   tokenWithAntiquot (unicodeSymbolNoAntiquot sym asciiSym)
15352
15353 /--
15354   Define parser for `$$e` (if anonymous == true) and `$$e:name`. Both
15355   forms can also be used with an appended `*` to turn them into an
15356   antiquotation "splice". If `kind` is given, it will additionally be checked
15357   when evaluating `match_syntax`. Antiquotations can be escaped as in `$$e`, which
15358   produces the syntax tree for `$$e`. -/
15359 def mkAntiquot (name : String) (kind : Option SyntaxNodeKind) (anonymous := true) : Parser :=
15360   let kind := (kind.getD Name.anonymous) ++ `antiquot
15361   let nameP := node `antiquotName $ checkNoWsBefore ("no space before ':'" ++ name ++ "'") >> symbol ":" >> nonReservedSymbol name
15362   -- if parsing the kind fails and `anonymous` is true, check that we're not ignoring a different
15363   -- antiquotation kind via `noImmediateColon`
15364   let nameP := if anonymous then nameP <|> checkNoImmediateColon >> pushNone else nameP
15365   -- antiquotations are not part of the "standard" syntax, so hide "expected '$'" on error
15366   leadingNode kind maxPrec $ atomic $
15367     setExpected [] "$" >>
15368     manyNoAntiquot (checkNoWsBefore "" >> "$") >>

```

```

15369     checkNoWsBefore "no space before spliced term" >> antiquotExpr >>
15370     nameP
15371
15372 def tryAnti (c : ParserContext) (s : ParserState) : Bool :=
15373   let (s, stx) := peekToken c s
15374   match stx with
15375   | Except.ok stx@(Syntax.atom _ sym) => sym == "$"
15376   | _                               => false
15377
15378 @[inline] def withAntiquotFn (antiquotP p : ParserFn) : ParserFn := fun c s =>
15379   if tryAnti c s then orelseFn antiquotP p c s else p c s
15380
15381 /-- Optimized version of `mkAntiquot ... <|> p`. -/
15382 @[inline] def withAntiquot (antiquotP p : Parser) : Parser := {
15383   fn := withAntiquotFn antiquotP.fn p.fn,
15384   info := orelseInfo antiquotP.info p.info
15385 }
15386
15387 def withoutInfo (p : Parser) : Parser :=
15388   { fn := p.fn }
15389
15390 /-- Parse `${p}suffix`, e.g. `${p},*`. -/
15391 def mkAntiquotSplice (kind : SyntaxNodeKind) (p suffix : Parser) : Parser :=
15392   let kind := kind ++ `antiquot_scope
15393   leadingNode kind maxPrec $ atomic $
15394     setExpected [] "$" >>
15395     manyNoAntiquot (checkNoWsBefore "" >> "$") >>
15396     checkNoWsBefore "no space before spliced term" >> symbol "[" >> node nullKind p >> symbol "]" >>
15397     suffix
15398
15399 @[inline] def withAntiquotSuffixSpliceFn (kind : SyntaxNodeKind) (p suffix : ParserFn) : ParserFn := fun c s => do
15400   let s := p c s
15401   if s.hasError || !s.stxStack.back.isAntiquot then
15402     return s
15403   let iniSz := s.stackSize
15404   let iniPos := s.pos
15405   let s := suffix c s
15406   if s.hasError then
15407     return s.restore iniSz iniPos
15408   s.mkNode (kind ++ `antiquot_suffix_splice) (s.stxStack.size - 2)
15409
15410 /-- Parse `suffix` after an antiquotation, e.g. `${x},*`, and put both into a new node. -/
15411 @[inline] def withAntiquotSuffixSplice (kind : SyntaxNodeKind) (p suffix : Parser) : Parser := {
15412   info := andthenInfo p.info suffix.info,
15413   fn := withAntiquotSuffixSpliceFn kind p.fn suffix.fn
15414 }
15415

```



```

15416 def withAntiquotSpliceAndSuffix (kind : SyntaxNodeKind) (p suffix : Parser) :=
15417   -- prevent `p`'s info from being collected twice
15418   withAntiquot (mkAntiquotSplice kind (withoutInfo p) suffix) (withAntiquotSuffixSplice kind p suffix)
15419
15420 def nodeWithAntiquot (name : String) (kind : SyntaxNodeKind) (p : Parser) (anonymous := false) : Parser :=
15421   withAntiquot (mkAntiquot name kind anonymous) $ node kind p
15422
15423 /- ===== -/
15424 /- End of Antiquotations -/
15425 /- ===== -/
15426
15427 def sepByElemParser (p : Parser) (sep : String) : Parser :=
15428   withAntiquotSpliceAndSuffix `sepBy p (symbol (sep.trim ++ " "))
15429
15430 def sepBy (p : Parser) (sep : String) (psep : Parser := symbol sep) (allowTrailingSep : Bool := false) : Parser :=
15431   sepByNoAntiquot (sepByElemParser p sep) psep allowTrailingSep
15432
15433 def sepBy1 (p : Parser) (sep : String) (psep : Parser := symbol sep) (allowTrailingSep : Bool := false) : Parser :=
15434   sepBy1NoAntiquot (sepByElemParser p sep) psep allowTrailingSep
15435
15436 def categoryParserOfStackFn (offset : Nat) : ParserFn := fun ctx s =>
15437   let stack := s.stxStack
15438   if stack.size < offset + 1 then
15439     s.mkUnexpectedError ("failed to determine parser category using syntax stack, stack is too small")
15440   else
15441     match stack.get! (stack.size - offset - 1) with
15442     | Syntax.ident __ catName _ => categoryParserFn catName ctx s
15443     | _ => s.mkUnexpectedError ("failed to determine parser category using syntax stack, the specified element on the stack is not an i
15444
15445 def categoryParserOfStack (offset : Nat) (prec : Nat := 0) : Parser :=
15446   { fn := fun c s => categoryParserOfStackFn offset { c with prec := prec } s }
15447
15448 unsafe def evalParserConstUnsafe (declName : Name) : ParserFn := fun ctx s =>
15449   match ctx.env.evalConstCheck Parser ctx.options `Lean.Parser.Parser declName <|>
15450     ctx.env.evalConstCheck Parser ctx.options `Lean.Parser.TrailingParser declName with
15451   | Except.ok p    => p.fn ctx s
15452   | Except.error e => s.mkUnexpectedError s!"error running parser {declName}: {e}"
15453
15454 @[implementedBy evalParserConstUnsafe]
15455 constant evalParserConst (declName : Name) : ParserFn
15456
15457 unsafe def parserOfStackFnUnsafe (offset : Nat) : ParserFn := fun ctx s =>
15458   let stack := s.stxStack
15459   if stack.size < offset + 1 then
15460     s.mkUnexpectedError ("failed to determine parser using syntax stack, stack is too small")
15461   else
15462     match stack.get! (stack.size - offset - 1) with

```

```

15463 | Syntax.ident (val := parserName) .. =>
15464   match ctx.resolveName parserName with
15465   | [(parserName, [])] =>
15466     let iniSz := s.stackSize
15467     let s := evalParserConst parserName ctx s
15468     if !s.hasError && s.stackSize != iniSz + 1 then
15469       s.mkUnexpectedError "expected parser to return exactly one syntax object"
15470     else
15471       s
15472   | _:::_ => s.mkUnexpectedError s!"ambiguous parser name {parserName}"
15473   | _ => s.mkUnexpectedError s!"unknown parser {parserName}"
15474   | _ => s.mkUnexpectedError ("failed to determine parser using syntax stack, the specified element on the stack is not an identifier"
15475
15476 @[implementedBy parserOfStackFnUnsafe]
15477 constant parserOfStackFn (offset : Nat) : ParserFn
15478
15479 def parserOfStack (offset : Nat) (prec : Nat := 0) : Parser :=
15480   { fn := fun c s => parserOfStackFn offset { c with prec := prec } s }
15481
15482 /- Run `declName` if possible and inside a quotation, or else `p`. The `ParserInfo` will always be taken from `p`. -/
15483 def evalInsideQuot (declName : Name) (p : Parser) : Parser := { p with
15484   fn := fun c s =>
15485     if c.insideQuot && c.env.contains declName then
15486       evalParserConst declName c s
15487     else
15488       p.fn c s }
15489
15490 private def mkResult (s : ParserState) (iniSz : Nat) : ParserState :=
15491   if s.stackSize == iniSz + 1 then s
15492   else s.mkNode nullKind iniSz -- throw error instead?
15493
15494 def leadingParserAux (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) : ParserFn := fun c s => do
15495   let iniSz := s.stackSize
15496   let (s, ps) := indexed tables.leadingTable c s behavior
15497   if s.hasError then
15498     return s
15499   let ps := tables.leadingParsers ++ ps
15500   if ps.isEmpty then
15501     return s.mkError (toString kind)
15502   let s := longestMatchFn none ps c s
15503   mkResult s iniSz
15504
15505 @[inline] def leadingParser (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) (antiquotParser : ParserFn) :
15506   withAntiquotFn antiquotParser (leadingParserAux kind tables behavior)
15507
15508 def trailingLoopStep (tables : PrattParsingTables) (left : Syntax) (ps : List (Parser × Nat)) : ParserFn := fun c s =>
15509   longestMatchFn left (ps ++ tables.trailingParsers) c s

```

```

15510
15511 partial def trailingLoop (tables : PrattParsingTables) (c : ParserContext) (s : ParserState) : ParserState := do
15512   let iniSz := s.stackSize
15513   let iniPos := s.pos
15514   let (s, ps) := indexed tables.trailingTable c s LeadingIdentBehavior.default
15515   if s.hasError then
15516     -- Discard token parse errors and break the trailing loop instead.
15517     -- The error will be flagged when the next leading position is parsed, unless the token
15518     -- is in fact valid there (e.g. EOI at command level, no-longer forbidden token)
15519     return s.restore iniSz iniPos
15520   if ps.isEmpty && tables.trailingParsers.isEmpty then
15521     return s -- no available trailing parser
15522   let left := s.stxStack.back
15523   let s := s.popSyntax
15524   let s := trailingLoopStep tables left ps c s
15525   if s.hasError then
15526     -- Discard non-consuming parse errors and break the trailing loop instead, restoring `left`.
15527     -- This is necessary for fallback parsers like `app` that pretend to be always applicable.
15528     return if s.pos == iniPos then s.restore (iniSz - 1) iniPos |>.pushSyntax left else s
15529   trailingLoop tables c s
15530
15531 /--
15532
15533 Implements a variant of Pratt's algorithm. In Pratt's algorithms tokens have a right and left binding power.
15534 In our implementation, parsers have precedence instead. This method selects a parser (or more, via
15535 `longestMatchFn`) from `leadingTable` based on the current token. Note that the unindexed `leadingParsers` parsers
15536 are also tried. We have the unindexed `leadingParsers` because some parsers do not have a "first token". Example:
15537 ```
15538 syntax term:51 "≤" ident "<" term "|" term : index
15539 ```
15540 Example, in principle, the set of first tokens for this parser is any token that can start a term, but this set
15541 is always changing. Thus, this parsing rule is stored as an unindexed leading parser at `leadingParsers`.
15542 After processing the leading parser, we chain with parsers from `trailingTable`/`trailingParsers` that have precedence
15543 at least `c.prec` where `c` is the `ParsingContext`. Recall that `c.prec` is set by `categoryParser`.
15544
15545 Note that in the original Pratt's algorithm, precedences are only checked before calling trailing parsers. In our
15546 implementation, leading *and* trailing parsers check the precedence. We claim our algorithm is more flexible,
15547 modular and easier to understand.
15548
15549 `antiquotParser` should be a `mkAntiquot` parser (or always fail) and is tried before all other parsers.
15550 It should not be added to the regular leading parsers because it would heavily
15551 overlap with antiquotation parsers nested inside them. -/
15552 @[inline] def prattParser (kind : Name) (tables : PrattParsingTables) (behavior : LeadingIdentBehavior) (antiquotParser : ParserFn) : P
15553   let iniSz := s.stackSize
15554   let iniPos := s.pos
15555   let s := leadingParser kind tables behavior antiquotParser c s
15556   if s.hasError then

```

```

15557     s
15558   else
15559     trailingLoop tables c s
15560
15561 def fieldIdxFn : ParserFn := fun c s =>
15562   let initStackSz := s.stackSize
15563   let iniPos := s.pos
15564   let curr := c.input.get iniPos
15565   if curr.isDigit && curr != '0' then
15566     let s := takeWhileFn (fun c => c.isDigit) c s
15567     mkNodeToken fieldIdxKind iniPos c s
15568   else
15569     s.mkErrorAt "field index" iniPos initStackSz
15570
15571 @[inline] def fieldIdx : Parser :=
15572   withAntiquot (mkAntiquot "fieldIdx" `fieldIdx) {
15573     fn := fieldIdxFn,
15574     info := mkAtomicInfo "fieldIdx"
15575   }
15576
15577 @[inline] def skip : Parser := {
15578   fn := fun c s => s,
15579   info := epsilonInfo
15580 }
15581
15582 end Parser
15583
15584 namespace Syntax
15585
15586 section
15587 variable { $\beta$  : Type} {m : Type → Type} [Monad m]
15588
15589 @[inline] def foldArgsM (s : Syntax) (f : Syntax →  $\beta$  → m  $\beta$ ) (b :  $\beta$ ) : m  $\beta$  :=
15590   s.getArgs.foldlM (flip f) b
15591
15592 @[inline] def foldArgs (s : Syntax) (f : Syntax →  $\beta$  →  $\beta$ ) (b :  $\beta$ ) :  $\beta$  :=
15593   Id.run (s.foldArgsM f b)
15594
15595 @[inline] def forArgsM (s : Syntax) (f : Syntax → m Unit) : m Unit :=
15596   s.foldArgsM (fun s _ => f s) ()
15597 end
15598
15599 end Syntax
15600 end Lean
15601 ::::::::::::::
15602 Parser/Command.lean
15603 ::::::::::::::

```

```

15604 /-
15605 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
15606 Released under Apache 2.0 license as described in the file LICENSE.
15607 Authors: Leonardo de Moura, Sebastian Ullrich
15608 -/
15609 import Lean.Parser.Term
15610 import Lean.Parser.Do
15611
15612 namespace Lean
15613 namespace Parser
15614
15615 /--
15616   Syntax quotation for terms and (lists of) commands. We prefer terms, so ambiguous quotations like
15617   `($x $y) will be parsed as an application, not two commands. Use `($x:command $y:command) instead.
15618   Multiple command will be put in a `null node, but a single command will not (so that you can directly
15619   match against a quotation in a command kind's elaborator). -/
15620 -- TODO: use two separate quotation parsers with parser priorities instead
15621 @[builtinTermParser] def Term.quot := leading_parser "`(" >> toggleInsideQuot (termParser <|> many1Unbox commandParser) >> ")"
15622
15623 namespace Command
15624
15625 def namedPrio := leading_parser (atomic "(" >> nonReservedSymbol "priority") >> " := " >> priorityParser >> ")")
15626 def optNamedPrio := optional namedPrio
15627
15628 def «private» := leading_parser "private "
15629 def «protected» := leading_parser "protected "
15630 def «visibility» := «private» <|> «protected»
15631 def «noncomputable» := leading_parser "noncomputable "
15632 def «unsafe» := leading_parser "unsafe "
15633 def «partial» := leading_parser "partial "
15634 def declModifiers (inline : Bool) := leading_parser optional docComment >> optional (Term.«attributes» >> if inline then skip else ppDe
15635 def declId := leading_parser ident >> optional (".{" >> sepBy1 ident ", " >> "}")
15636 def declSig := leading_parser many (ppSpace >> (Term.simpleBinderWithoutType <|> Term.bracketedBinder)) >> Term.typeSpec
15637 def optDeclSig := leading_parser many (ppSpace >> (Term.simpleBinderWithoutType <|> Term.bracketedBinder)) >> Term.optType
15638 def declValSimple := leading_parser " :=\n" >> termParser >> optional Term.whereDecls
15639 def declValEqns := leading_parser Term.matchAltsWhereDecls
15640 def declVal := declValSimple <|> declValEqns <|> Term.whereDecls
15641 def «abbrev» := leading_parser "abbrev " >> declId >> optDeclSig >> declVal
15642 def «def» := leading_parser "def " >> declId >> optDeclSig >> declVal
15643 def «theorem» := leading_parser "theorem " >> declId >> declSig >> declVal
15644 def «constant» := leading_parser "constant " >> declId >> declSig >> optional declValSimple
15645 def «instance» := leading_parser Term.attrKind >> "instance " >> optNamedPrio >> optional declId >> declSig >> declVal
15646 def «axiom» := leading_parser "axiom " >> declId >> declSig
15647 def «example» := leading_parser "example " >> declSig >> declVal
15648 def inferMod := leading_parser atomic (symbol "{" >> "}")
15649 def ctor := leading_parser "\n| " >> declModifiers true >> ident >> optional inferMod >> optDeclSig
15650 def optDeriving := leading_parser optional (atomic ("deriving " >> notSymbol "instance") >> sepBy1 ident ", ")

```

```

15651 def «inductive»      := leading_parser "inductive " >> declId >> optDeclSig >> optional (symbol ":@" <|> "where") >> many ctor >> optDe
15652 def classInductive    := leading_parser atomic (group (symbol "class " >> "inductive ")) >> declId >> optDeclSig >> optional (symbol ":@"
15653 def structExplicitBinder := leading_parser atomic (declModifiers true >> "(") >> many1 ident >> optional inferMod >> optDeclSig >> opti
15654 def structImplicitBinder := leading_parser atomic (declModifiers true >> "{") >> many1 ident >> optional inferMod >> declSig >> "}"
15655 def structInstBinder    := leading_parser atomic (declModifiers true >> "[") >> many1 ident >> optional inferMod >> declSig >> "]"
15656 def structSimpleBinder  := leading_parser atomic (declModifiers true >> ident) >> optional inferMod >> optDeclSig >> optional Term.bin
15657 def structFields       := leading_parser many1 indent (ppLine >> checkColGe >> (structExplicitBinder <|> structImplicitBinder <|> struct
15658 def structCtor          := leading_parser atomic (declModifiers true >> ident >> optional inferMod >> " :: ")
15659 def structureTk         := leading_parser "structure "
15660 def classTk             := leading_parser "class "
15661 def «extends»          := leading_parser " extends " >> sepBy1 termParser ", "
15662 def «structure»        := leading_parser
15663   (structureTk <|> classTk) >> declId >> many Term.bracketedBinder >> optional «extends» >> Term.optType
15664   >> optional ((symbol ":@" <|> " where ") >> optional structCtor >> structFields)
15665   >> optDeriving
15666 @[builtinCommandParser] def declaration := leading_parser
15667 declModifiers false >> («abbrev» <|> «def» <|> «theorem» <|> «constant» <|> «instance» <|> «axiom» <|> «example» <|> «inductive» <|> cl
15668 @[builtinCommandParser] def «deriving» := leading_parser "deriving " >> "instance " >> sepBy1 ident ", " >> " for " >> sepBy1 ident
15669 @[builtinCommandParser] def «section»   := leading_parser "section " >> optional ident
15670 @[builtinCommandParser] def «namespace» := leading_parser "namespace " >> ident
15671 @[builtinCommandParser] def «end»       := leading_parser "end " >> optional ident
15672 @[builtinCommandParser] def «variable»  := leading_parser "variable" >> many1 Term.bracketedBinder
15673 @[builtinCommandParser] def «universe»  := leading_parser "universe " >> ident
15674 @[builtinCommandParser] def «universes» := leading_parser "universes " >> many1 ident
15675 @[builtinCommandParser] def check       := leading_parser "#check " >> termParser
15676 @[builtinCommandParser] def check_failure := leading_parser "#check_failure " >> termParser -- Like `#check`, but succeeds only if ter
15677 @[builtinCommandParser] def reduce     := leading_parser "#reduce " >> termParser
15678 @[builtinCommandParser] def eval       := leading_parser "#eval " >> termParser
15679 @[builtinCommandParser] def synth      := leading_parser "#synth " >> termParser
15680 @[builtinCommandParser] def exit       := leading_parser "#exit"
15681 @[builtinCommandParser] def print      := leading_parser "#print " >> (ident <|> strLit)
15682 @[builtinCommandParser] def printAxioms := leading_parser "#print " >> nonReservedSymbol "axioms " >> ident
15683 @[builtinCommandParser] def «resolve_name» := leading_parser "#resolve_name " >> ident
15684 @[builtinCommandParser] def «init_quot» := leading_parser "init_quot"
15685 def optionValue := nonReservedSymbol "true" <|> nonReservedSymbol "false" <|> strLit <|> numLit
15686 @[builtinCommandParser] def «set_option» := leading_parser "set_option " >> ident >> ppSpace >> optionValue
15687 def eraseAttr := leading_parser "-" >> ident
15688 @[builtinCommandParser] def «attribute» := leading_parser "attribute " >> "[" >> sepBy1 (eraseAttr <|> Term.attrInstance) ", " >> "]"
15689 @[builtinCommandParser] def «export»    := leading_parser "export " >> ident >> "(" >> many1 ident >> ")"
15690 def openHiding      := leading_parser atomic (ident >> "hiding") >> many1 ident
15691 def openRenamingItem := leading_parser ident >> unicodeSymbol "→" "->" >> ident
15692 def openRenaming     := leading_parser atomic (ident >> "renaming") >> sepBy1 openRenamingItem ", "
15693 def openOnly         := leading_parser atomic (ident >> "(") >> many1 ident >> ")"
15694 def openSimple       := leading_parser many1 ident
15695 def openDecl         := openHiding <|> openRenaming <|> openOnly <|> openSimple
15696 @[builtinCommandParser] def «open»      := leading_parser "open " >> openDecl
15697

```

```

15698 @[builtinCommandParser] def «mutual» := leading_parser "mutual " >> many1 (ppLine >> notSymbol "end" >> commandParser) >> ppDedent (ppL
15699 @[builtinCommandParser] def «initialize» := leading_parser "initialize " >> optional (atomic (ident >> Term.typeSpec >> Term.leftArrow)
15700 @[builtinCommandParser] def «builtin_initialize» := leading_parser "builtin_initialize " >> optional (atomic (ident >> Term.typeSpec >>
15701
15702 @[builtinCommandParser] def «in» := trailing_parser " in " >> commandParser
15703
15704 @[runBuiltinParserAttributeHooks] abbrev declModifiersF := declModifiers false
15705 @[runBuiltinParserAttributeHooks] abbrev declModifiersT := declModifiers true
15706
15707 builtin_initialize
15708   register_parser_alias "declModifiers"      declModifiersF
15709   register_parser_alias "nestedDeclModifiers" declModifiersT
15710   register_parser_alias "declId"             declId
15711   register_parser_alias "declSig"            declSig
15712   register_parser_alias "declVal"            declVal
15713   register_parser_alias "optDeclSig"          optDeclSig
15714   register_parser_alias "openDecl"           openDecl
15715
15716 end Command
15717
15718 namespace Term
15719 @[builtinTermParser] def «open» := leading_parser:leadPrec "open " >> Command.openDecl >> " in " >> termParser
15720 @[builtinTermParser] def «set_option» := leading_parser:leadPrec "set_option " >> ident >> ppSpace >> Command.optionValue >> " in " >>
15721 end Term
15722
15723 namespace Tactic
15724 @[builtinTacticParser] def «open» := leading_parser:leadPrec "open " >> Command.openDecl >> " in " >> tacticSeq
15725 @[builtinTacticParser] def «set_option» := leading_parser:leadPrec "set_option " >> ident >> ppSpace >> Command.optionValue >> " in " >
15726 end Tactic
15727
15728 end Parser
15729 end Lean
15730 ::::::::::::::
15731 Parser/Do.lean
15732 ::::::::::::::
15733 /-
15734 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
15735 Released under Apache 2.0 license as described in the file LICENSE.
15736 Authors: Leonardo de Moura
15737 -/
15738 import Lean.Parser.Term
15739
15740 namespace Lean
15741 namespace Parser
15742
15743 builtin_initialize registerBuiltinParserAttribute `builtinDoElemParser `doElem
15744 builtin_initialize registerBuiltinDynamicParserAttribute `doElemParser `doElem

```



```

15745
15746 @[inline] def doElemParser (rbp : Nat := 0) : Parser :=
15747   categoryParser `doElem rbp
15748
15749 namespace Term
15750 def leftArrow : Parser := unicodeSymbol " ← " <- "
15751 @[builtinTermParser] def liftMethod := leading_parser.minPrec leftArrow >> termParser
15752
15753 def doSeqItem      := leading_parser ppLine >> doElemParser >> optional ";"
15754 def doSeqIndent   := leading_parser many1Indent doSeqItem
15755 def doSeqBracketed := leading_parser "{" >> withoutPosition (many1 doSeqItem) >> ppLine >> "}"
15756 def doSeq         := doSeqBracketed <|> doSeqIndent
15757
15758 def termBeforeDo := withForbidden "do" termParser
15759
15760 attribute [runBuiltinParserAttributeHooks] doSeq termBeforeDo
15761
15762 builtin_initialize
15763   register_parser_alias "doSeq" doSeq
15764   register_parser_alias "termBeforeDo" termBeforeDo
15765
15766 def notFollowedByRedefinedTermToken :=
15767   -- 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
15768   -- "open" command following `do`-block. If we don't add `do`, then users would have to indent `do` blocks or use `{ ... }`.
15769   notFollowedBy ("set_option" <|> "open" <|> "if" <|> "match" <|> "let" <|> "have" <|> "do" <|> "dbg_trace" <|> "assert!" <|> "for" <|>
15770
15771 @[builtinDoElemParser] def doLet      := leading_parser "let " >> optional "mut " >> letDecl
15772
15773 @[builtinDoElemParser] def doLetRec  := leading_parser group ("let " >> nonReservedSymbol "rec ") >> letRecDecls
15774 def doIdDecl   := leading_parser atomic (ident >> optType >> leftArrow) >> doElemParser
15775 def doPatDecl  := leading_parser atomic (termParser >> leftArrow) >> doElemParser >> optional (checkColGt >> " | " >> doElemParser)
15776 @[builtinDoElemParser] def doLetArrow := leading_parser withPosition ("let " >> optional "mut " >> (doIdDecl <|> doPatDecl))
15777
15778 -- We use `letIdDeclNoBinders` to define `doReassign`.
15779 -- Motivation: we do not reassign functions, and avoid parser conflict
15780 def letIdDeclNoBinders := node `Lean.Parser.Term.letIdDecl $ atomic (ident >> pushNone >> optType >> " := ") >> termParser
15781
15782 @[builtinDoElemParser] def doReassign      := leading_parser notFollowedByRedefinedTermToken >> (letIdDeclNoBinders <|> letPatDecl)
15783 @[builtinDoElemParser] def doReassignArrow := leading_parser notFollowedByRedefinedTermToken >> withPosition (doIdDecl <|> doPatDecl)
15784 @[builtinDoElemParser] def doHave        := leading_parser "have " >> Term.haveDecl
15785 /-
15786 In `do` blocks, we support `if` without an `else`. Thus, we use indentation to prevent examples such as
15787 ```
15788 if c_1 then
15789   if c_2 then
15790     action_1
15791 else

```



```

15792   action_2
15793   ``
15794   from being parsed as
15795   ``
15796   if c_1 then {
15797     if c_2 then {
15798       action_1
15799     } else {
15800       action_2
15801     }
15802   }
15803   ``
15804   We also have special support for `else if` because we don't want to write
15805   ``
15806   if c_1 then
15807     action_1
15808   else if c_2 then
15809     action_2
15810   else
15811     action_3
15812   ``
15813   -/
15814   def elseIf := atomic (group (withPosition (" else " >> checkLineEq >> " if ")))
15815   -- ensure `if $e then ...` still binds to `e:term`
15816   def doIfLetPure := leading_parser " := " >> termParser
15817   def doIfLetBind := leading_parser " ← " >> termParser
15818   def doIfLet     := nodeWithAntiquot "doIfLet"      `Lean.Parser.Term.doIfLet      <| "let " >> termParser >> (doIfLetPure <|> doIfLetBind)
15819   def doIfProp    := nodeWithAntiquot "doIfProp"    `Lean.Parser.Term.doIfProp    <| optIdent >> termParser
15820   def doIfCond    := withAntiquot (mkAntiquot "doIfCond" none (anonymous := false)) <| doIfLet <|> doIfProp
15821   @[builtinDoElemParser] def doIf := leading_parser withPosition $
15822     "if " >> doIfCond >> " then " >> doSeq
15823     >> many (checkColGe "'else if' in 'do' must be indented" >> group (elseIf >> doIfCond >> " then " >> doSeq))
15824     >> optional (checkColGe "'else' in 'do' must be indented" >> " else " >> doSeq)
15825   @[builtinDoElemParser] def doUnless := leading_parser "unless " >> withForbidden "do" termParser >> "do " >> doSeq
15826   def doForDecl := leading_parser termParser >> " in " >> withForbidden "do" termParser
15827   @[builtinDoElemParser] def doFor   := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq
15828
15829   def doMatchAlts := matchAlts (rhsParser := doSeq)
15830   @[builtinDoElemParser] def doMatch := leading_parser:leadPrec "match " >> optional Term.generalizingParam >> sepBy1 matchDiscr ", " >> doSeq
15831
15832   def doCatch      := leading_parser atomic ("catch " >> binderIdent) >> optional (" : " >> termParser) >> darrow >> doSeq
15833   def doCatchMatch := leading_parser "catch " >> doMatchAlts
15834   def doFinally    := leading_parser "finally " >> doSeq
15835   @[builtinDoElemParser] def doTry   := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally
15836
15837   @[builtinDoElemParser] def doBreak   := leading_parser "break"
15838   @[builtinDoElemParser] def doContinue := leading_parser "continue"

```

```

15839 @[builtinDoElemParser] def doReturn      := leading_parser:leadPrec withPosition ("return " >> optional (checkLineEq >> termParser))
15840 @[builtinDoElemParser] def doDbgTrace    := leading_parser:leadPrec "dbg_trace " >> ((interpolatedStr termParser) <|> termParser)
15841 @[builtinDoElemParser] def doAssert      := leading_parser:leadPrec "assert! " >> termParser
15842
15843 /-
15844 We use `notFollowedBy` to avoid counterintuitive behavior.
15845
15846 For example, the `if`-term parser
15847 doesn't enforce indentation restrictions, but we don't want it to be used when `doIf` fails.
15848 Note that parser priorities would not solve this problem since the `doIf` parser is failing while the `if`
15849 parser is succeeding. The first `notFollowedBy` prevents this problem.
15850
15851 Consider the `doElem` `x := (a, b)` it contains an error since we are using `)` instead of `)`. Thus, `doReassign` parser fails.
15852 However, `doExpr` would succeed consuming just `x`, and cryptic error message is generated after that.
15853 The second `notFollowedBy` prevents this problem.
15854 -/
15855 @[builtinDoElemParser] def doExpr      := leading_parser notFollowedByRedefinedTermToken >> termParser >> notFollowedBy (symbol ":@" <|> s
15856 @[builtinDoElemParser] def doNested := leading_parser "do " >> doSeq
15857
15858 @[builtinTermParser] def «do» := leading_parser:argPrec "do " >> doSeq
15859
15860 @[builtinTermParser] def doElem.quot : Parser := leading_parser "`(doElem|" >> toggleInsideQuot doElemParser >> ")"
15861
15862 /- macros for using `unless`, `for`, `try`, `return` as terms. They expand into `do unless ...`, `do for ...`, `do try ...`, and `do re
15863 @[builtinTermParser] def termUnless := leading_parser "unless " >> withForbidden "do" termParser >> "do " >> doSeq
15864 @[builtinTermParser] def termFor    := leading_parser "for " >> sepBy1 doForDecl ", " >> "do " >> doSeq
15865 @[builtinTermParser] def termTry    := leading_parser "try " >> doSeq >> many (doCatch <|> doCatchMatch) >> optional doFinally
15866 @[builtinTermParser] def termReturn := leading_parser:leadPrec withPosition ("return " >> optional (checkLineEq >> termParser))
15867
15868 end Term
15869 end Parser
15870 end Lean
15871 ::::::::::::::
15872 Parser/Extension.lean
15873 ::::::::::::::
15874 /-
15875 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
15876 Released under Apache 2.0 license as described in the file LICENSE.
15877 Authors: Leonardo de Moura, Sebastian Ullrich
15878 -/
15879 import Lean.ScopedEnvExtension
15880 import Lean.Parser.Basic
15881 import Lean.Parser.StrInterpolation
15882 import Lean.KeyedDeclsAttribute
15883
15884 /-! Extensible parsing via attributes -/
15885

```

```

15886 namespace Lean
15887 namespace Parser
15888
15889 builtin_initialize builtinTokenTable : IO.Ref TokenTable ← IO.mkRef {}
15890
15891 /- Global table with all SyntaxNodeKind's -/
15892 builtin_initialize builtinSyntaxNodeKindSetRef : IO.Ref SyntaxNodeKindSet ← IO.mkRef {}
15893
15894 def registerBuiltinNodeKind (k : SyntaxNodeKind) : IO Unit :=
15895   builtinSyntaxNodeKindSetRef.modify fun s => s.insert k
15896
15897 builtin_initialize
15898   registerBuiltinNodeKind choiceKind
15899   registerBuiltinNodeKind identKind
15900   registerBuiltinNodeKind strLitKind
15901   registerBuiltinNodeKind numLitKind
15902   registerBuiltinNodeKind scientificLitKind
15903   registerBuiltinNodeKind charLitKind
15904   registerBuiltinNodeKind nameLitKind
15905
15906 builtin_initialize builtinParserCategoriesRef : IO.Ref ParserCategories ← IO.mkRef {}
15907
15908 private def throwParserCategoryAlreadyDefined {α} (catName : Name) : ExceptT String Id α :=
15909   throw s!"parser category '{catName}' has already been defined"
15910
15911 private def addParserCategoryCore (categories : ParserCategories) (catName : Name) (initial : ParserCategory) : Except String ParserCat
15912   if categories.contains catName then
15913     throwParserCategoryAlreadyDefined catName
15914   else
15915     pure $ categories.insert catName initial
15916
15917 /-- All builtin parser categories are Pratt's parsers -/
15918
15919 private def addBuiltinParserCategory (catName : Name) (behavior : LeadingIdentBehavior) : IO Unit := do
15920   let categories ← builtinParserCategoriesRef.get
15921   let categories ← IO.ofExcept $ addParserCategoryCore categories catName { tables := {}, behavior := behavior }
15922   builtinParserCategoriesRef.set categories
15923
15924 namespace ParserExtension
15925
15926 inductive OLeanEntry where
15927   | token      (val : Token) : OLeanEntry
15928   | kind       (val : SyntaxNodeKind) : OLeanEntry
15929   | category   (catName : Name) (behavior : LeadingIdentBehavior)
15930   | parser     (catName : Name) (declName : Name) (prio : Nat) : OLeanEntry
15931   deriving Inhabited
15932

```

```

15933 inductive Entry where
15934   | token      (val : Token) : Entry
15935   | kind       (val : SyntaxNodeKind) : Entry
15936   | category   (catName : Name) (behavior : LeadingIdentBehavior)
15937   | parser     (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : Entry
15938   deriving Inhabited
15939
15940 def Entry.to0LeanEntry : Entry → 0LeanEntry
15941   | token v          => 0LeanEntry.token v
15942   | kind v           => 0LeanEntry.kind v
15943   | category c b     => 0LeanEntry.category c b
15944   | parser c d _ _ prio => 0LeanEntry.parser c d prio
15945
15946 structure State where
15947   tokens      : TokenTable := {}
15948   kinds       : SyntaxNodeKindSet := {}
15949   categories   : ParserCategories := {}
15950   deriving Inhabited
15951
15952 end ParserExtension
15953
15954 open ParserExtension in
15955 abbrev ParserExtension := ScopedEnvExtension 0LeanEntry Entry State
15956
15957 private def ParserExtension.mkInitial : IO ParserExtension.State := do
15958   let tokens      ← builtinTokenTable.get
15959   let kinds       ← builtinSyntaxNodeKindSetRef.get
15960   let categories ← builtinParserCategoriesRef.get
15961   pure { tokens := tokens, kinds := kinds, categories := categories }
15962
15963 private def addTokenConfig (tokens : TokenTable) (tk : Token) : Except String TokenTable := do
15964   if tk == "" then throw "invalid empty symbol"
15965   else match tokens.find? tk with
15966     | none  => pure $ tokens.insert tk tk
15967     | some _ => pure tokens
15968
15969 def throwUnknownParserCategory {α} (catName : Name) : ExceptT String Id α :=
15970   throw s!"unknown parser category '{catName}'"
15971
15972 abbrev getCategory (categories : ParserCategories) (catName : Name) : Option ParserCategory :=
15973   categories.find? catName
15974
15975 def addLeadingParser (categories : ParserCategories) (catName : Name) (parserName : Name) (p : Parser) (prio : Nat) : Except String Par
15976   match getCategory categories catName with
15977   | none  =>
15978     throwUnknownParserCategory catName
15979   | some cat =>

```

```

15980 let addTokens (tk : List Token) : Except String ParserCategories :=
15981   let tks := tks.map $ fun tk => Name.mkSimple tk
15982   let tables := tks.eraseDups.foldl (fun (tables : PrattParsingTables) tk => { tables with leadingTable := tables.leadingTable.inse
15983     pure $ categories.insert catName { cat with tables := tables }
15984   match p.info.firstTokens with
15985   | FirstTokens.tokens tks => addTokens tks
15986   | FirstTokens.optTokens tks => addTokens tks
15987   | _ =>
15988     let tables := { cat.tables with leadingParsers := (p, prio) :: cat.tables.leadingParsers }
15989     pure $ categories.insert catName { cat with tables := tables }
15990
15991 private def addTrailingParserAux (tables : PrattParsingTables) (p : TrailingParser) (prio : Nat) : PrattParsingTables :=
15992   let addTokens (tk : List Token) : PrattParsingTables :=
15993     let tks := tks.map fun tk => Name.mkSimple tk
15994     tks.eraseDups.foldl (fun (tables : PrattParsingTables) tk => { tables with trailingTable := tables.trailingTable.insert tk (p, prio
15995   match p.info.firstTokens with
15996   | FirstTokens.tokens tks => addTokens tks
15997   | FirstTokens.optTokens tks => addTokens tks
15998   | _ => { tables with trailingParsers := (p, prio) :: tables.trailingParsers }
15999
16000 def addTrailingParser (categories : ParserCategories) (catName : Name) (p : TrailingParser) (prio : Nat) : Except String ParserCategori
16001   match getCategory categories catName with
16002   | none => throwUnknownParserCategory catName
16003   | some cat => pure $ categories.insert catName { cat with tables := addTrailingParserAux cat.tables p prio }
16004
16005 def addParser (categories : ParserCategories) (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : Except St
16006   match leading, p with
16007   | true, p => addLeadingParser categories catName declName p prio
16008   | false, p => addTrailingParser categories catName p prio
16009
16010 def addParserTokens (tokenTable : TokenTable) (info : ParserInfo) : Except String TokenTable :=
16011   let newTokens := info.collectTokens []
16012   newTokens.foldlM addTokenConfig tokenTable
16013
16014 private def updateBuiltinTokens (info : ParserInfo) (declName : Name) : IO Unit := do
16015   let tokenTable ← builtinTokenTable.swap {}
16016   match addParserTokens tokenTable info with
16017   | Except.ok tokenTable => builtinTokenTable.set tokenTable
16018   | Except.error msg => throw (IO.userError s!"invalid builtin parser '{declName}', {msg}")
16019
16020 def addBuiltinParser (catName : Name) (declName : Name) (leading : Bool) (p : Parser) (prio : Nat) : IO Unit := do
16021   let p := evalInsideQuot declName p
16022   let categories ← builtinParserCategoriesRef.get
16023   let categories ← IO.ofExcept $ addParser categories catName declName leading p prio
16024   builtinParserCategoriesRef.set categories
16025   builtinSyntaxNodeKindSetRef.modify p.info.collectKinds
16026   updateBuiltinTokens p.info declName

```

```

16027
16028 def addBuiltinLeadingParser (catName : Name) (declName : Name) (p : Parser) (prio : Nat) : IO Unit :=
16029   addBuiltinParser catName declName true p prio
16030
16031 def addBuiltinTrailingParser (catName : Name) (declName : Name) (p : TrailingParser) (prio : Nat) : IO Unit :=
16032   addBuiltinParser catName declName false p prio
16033
16034 def ParserExtension.addEntryImpl (s : State) (e : Entry) : State :=
16035   match e with
16036   | Entry.token tk =>
16037     match addTokenConfig s.tokens tk with
16038     | Except.ok tokens => { s with tokens := tokens }
16039     | _                 => unreachable!
16040   | Entry.kind k =>
16041     { s with kinds := s.kinds.insert k }
16042   | Entry.category catName behavior =>
16043     if s.categories.contains catName then s
16044     else { s with
16045       categories := s.categories.insert catName { tables := {}, behavior := behavior } }
16046   | Entry.parser catName declName leading parser prio =>
16047     match addParser s.categories catName declName leading parser prio with
16048     | Except.ok categories => { s with categories := categories }
16049     | _ => unreachable!
16050
16051 unsafe def mkParserOfConstantUnsafe
16052   (categories : ParserCategories) (constName : Name) (compileParserDescr : ParserDescr → ImportM Parser) : ImportM (Bool × Parser) :=
16053   let env := (← read).env
16054   let opts := (← read).opts
16055   match env.find? constName with
16056   | none => throw !s!"unknown constant '{constName}'"
16057   | some info =>
16058     match info.type with
16059     | Expr.const `Lean.Parser.TrailingParser _ _ =>
16060       let p ← IO.ofExcept $ env.evalConst Parser opts constName
16061       pure (false, p)
16062     | Expr.const `Lean.Parser.Parser _ _ =>
16063       let p ← IO.ofExcept $ env.evalConst Parser opts constName
16064       pure (true, p)
16065     | Expr.const `Lean.ParserDescr _ _ =>
16066       let d ← IO.ofExcept $ env.evalConst ParserDescr opts constName
16067       let p ← compileParserDescr d
16068       pure (true, p)
16069     | Expr.const `Lean.TrailingParserDescr _ _ =>
16070       let d ← IO.ofExcept $ env.evalConst TrailingParserDescr opts constName
16071       let p ← compileParserDescr d
16072       pure (false, p)
16073   | _ => throw !s!"unexpected parser type at '{constName}' (`ParserDescr`, `TrailingParserDescr`, `Parser` or `TrailingParser` expect

```

```

16074
16075 @[implementedBy mkParserOfConstantUnsafe]
16076 constant mkParserOfConstantAux
16077   (categories : ParserCategories) (constName : Name) (compileParserDescr : ParserDescr → ImportM Parser) : ImportM (Bool × Parser)
16078
16079 /- Parser aliases for making `ParserDescr` extensible -/
16080 inductive AliasValue (α : Type) where
16081   | const (p : α)
16082   | unary (p : α → α)
16083   | binary (p : α → α → α)
16084
16085 abbrev AliasTable (α) := NameMap (AliasValue α)
16086
16087 def registerAliasCore {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) (value : AliasValue α) : IO Unit := do
16088   unless (← IO.initializing) do throw !s!"aliases can only be registered during initialization"
16089   if (← mapRef.get).contains aliasName then
16090     throw !s!"alias '{aliasName}' has already been declared"
16091   mapRef.modify (·.insert aliasName value)
16092
16093 def getAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (Option (AliasValue α)) := do
16094   return (← mapRef.get).find? aliasName
16095
16096 def getConstAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO α := do
16097   match (← getAlias mapRef aliasName) with
16098   | some (AliasValue.const v) => pure v
16099   | some (AliasValue.unary _) => throw !s!"parser '{aliasName}' is not a constant, it takes one argument"
16100   | some (AliasValue.binary _) => throw !s!"parser '{aliasName}' is not a constant, it takes two arguments"
16101   | none => throw !s!"parser '{aliasName}' was not found"
16102
16103 def getUnaryAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (α → α) := do
16104   match (← getAlias mapRef aliasName) with
16105   | some (AliasValue.unary v) => pure v
16106   | some _ => throw !s!"parser '{aliasName}' does not take one argument"
16107   | none => throw !s!"parser '{aliasName}' was not found"
16108
16109 def getBinaryAlias {α} (mapRef : IO.Ref (AliasTable α)) (aliasName : Name) : IO (α → α → α) := do
16110   match (← getAlias mapRef aliasName) with
16111   | some (AliasValue.binary v) => pure v
16112   | some _ => throw !s!"parser '{aliasName}' does not take two arguments"
16113   | none => throw !s!"parser '{aliasName}' was not found"
16114
16115 abbrev ParserAliasValue := AliasValue Parser
16116
16117 builtin_initialize parserAliasesRef : IO.Ref (NameMap ParserAliasValue) ← IO.mkRef {}
16118
16119 -- Later, we define macro registerParserAlias! which registers a parser, formatter and parenthesizer
16120 def registerAlias (aliasName : Name) (p : ParserAliasValue) : IO Unit := do

```

```

16121 registerAliasCore parserAliasesRef aliasName p
16122
16123 instance : Coe Parser ParserAliasValue := { coe := AliasValue.const }
16124 instance : Coe (Parser → Parser) ParserAliasValue := { coe := AliasValue.unary }
16125 instance : Coe (Parser → Parser → Parser) ParserAliasValue := { coe := AliasValue.binary }
16126
16127 def isParserAlias (aliasName : Name) : IO Bool := do
16128   match (← getAlias parserAliasesRef aliasName) with
16129   | some _ => pure true
16130   | _      => pure false
16131
16132 def ensureUnaryParserAlias (aliasName : Name) : IO Unit :=
16133   discard $ getUnaryAlias parserAliasesRef aliasName
16134
16135 def ensureBinaryParserAlias (aliasName : Name) : IO Unit :=
16136   discard $ getBinaryAlias parserAliasesRef aliasName
16137
16138 def ensureConstantParserAlias (aliasName : Name) : IO Unit :=
16139   discard $ getConstAlias parserAliasesRef aliasName
16140
16141 partial def compileParserDescr (categories : ParserCategories) (d : ParserDescr) : ImportM Parser :=
16142   let rec visit : ParserDescr → ImportM Parser
16143   | ParserDescr.const n                => getConstAlias parserAliasesRef n
16144   | ParserDescr.unary n d              => return (← getUnaryAlias parserAliasesRef n) (← visit d)
16145   | ParserDescr.binary n d₁ d₂        => return (← getBinaryAlias parserAliasesRef n) (← visit d₁) (← visit d₂)
16146   | ParserDescr.node k prec d         => return leadingNode k prec (← visit d)
16147   | ParserDescr.nodeWithAntiquot n k d => return nodeWithAntiquot n k (← visit d) (anonymous := true)
16148   | ParserDescr.sepBy p sep psep trail => return sepBy (← visit p) sep (← visit psep) trail
16149   | ParserDescr.sepBy1 p sep psep trail => return sepBy1 (← visit p) sep (← visit psep) trail
16150   | ParserDescr.trailingNode k prec lhsPrec d => return trailingNode k prec lhsPrec (← visit d)
16151   | ParserDescr.symbol tk              => return symbol tk
16152   | ParserDescr.nonReservedSymbol tk includeIdent => return nonReservedSymbol tk includeIdent
16153   | ParserDescr.parser constName       => do
16154     let (_, p) ← mkParserOfConstantAux categories constName visit;
16155     pure p
16156   | ParserDescr.cat catName prec       =>
16157     match getCategory categories catName with
16158     | some _ => pure $ categoryParser catName prec
16159     | none   => IO.ofExcept $ throwUnknownParserCategory catName
16160   visit d
16161
16162 def mkParserOfConstant (categories : ParserCategories) (constName : Name) : ImportM (Bool × Parser) :=
16163   mkParserOfConstantAux categories constName (compileParserDescr categories)
16164
16165 structure ParserAttributeHook where
16166   /- Called after a parser attribute is applied to a declaration. -/
16167   postAdd (catName : Name) (declName : Name) (builtin : Bool) : AttrM Unit

```



```

16168
16169 builtin_initialize parserAttributeHooks : IO.Ref (List ParserAttributeHook) ← IO.mkRef {}
16170
16171 def registerParserAttributeHook (hook : ParserAttributeHook) : IO Unit := do
16172   parserAttributeHooks.modify fun hooks => hook::hooks
16173
16174 def runParserAttributeHooks (catName : Name) (declName : Name) (builtin : Bool) : AttrM Unit := do
16175   let hooks ← parserAttributeHooks.get
16176   hooks.forM fun hook => hook.postAdd catName declName builtin
16177
16178 builtin_initialize
16179   registerBuiltinAttribute {
16180     name := `runBuiltinParserAttributeHooks,
16181     descr := "explicitly run hooks normally activated by builtin parser attributes",
16182     add := fun decl stx persistent => do
16183       Attribute.Builtin.ensureNoArgs stx
16184       runParserAttributeHooks Name.anonymous decl (builtin := true)
16185   }
16186
16187 builtin_initialize
16188   registerBuiltinAttribute {
16189     name := `runParserAttributeHooks,
16190     descr := "explicitly run hooks normally activated by parser attributes",
16191     add := fun decl stx persistent => do
16192       Attribute.Builtin.ensureNoArgs stx
16193       runParserAttributeHooks Name.anonymous decl (builtin := false)
16194   }
16195
16196 private def ParserExtension.OLeanEntry.toEntry (s : State) : OLeanEntry → ImportM Entry
16197 | token tk      => return Entry.token tk
16198 | kind k        => return Entry.kind k
16199 | category c l  => return Entry.category c l
16200 | parser catName declName prio => do
16201   let (leading, p) ← mkParserOfConstant s.categories declName
16202   Entry.parser catName declName leading p prio
16203
16204 builtin_initialize parserExtension : ParserExtension ←
16205   registerScopedEnvExtension {
16206     name      := `parserExt
16207     mkInitial := ParserExtension.mkInitial
16208     addEntry  := ParserExtension.addEntryImpl
16209     toOLeanEntry := ParserExtension.Entry.toOLeanEntry
16210     ofOLeanEntry := ParserExtension.OLeanEntry.toEntry
16211   }
16212
16213 def isParserCategory (env : Environment) (catName : Name) : Bool :=
16214   (parserExtension.getState env).categories.contains catName

```

```

16215
16216 def addParserCategory (env : Environment) (catName : Name) (behavior : LeadingIdentBehavior) : Except String Environment := do
16217   if isParserCategory env catName then
16218     throwParserCategoryAlreadyDefined catName
16219   else
16220     return parserExtension.addEntry env <| ParserExtension.Entry.category catName behavior
16221
16222 def leadingIdentBehavior (env : Environment) (catName : Name) : LeadingIdentBehavior :=
16223   match getCategory (parserExtension.getState env).categories catName with
16224   | none      => LeadingIdentBehavior.default
16225   | some cat => cat.behavior
16226
16227 def mkCategoryAntiquotParser (kind : Name) : Parser :=
16228   mkAntiquot kind.toString none
16229
16230 -- helper decl to work around inlining issue https://github.com/leanprover/lean4/commit/3f6de2af06dd9a25f62294129f64bc05a29ea912#r41340.
16231 @[inline] private def mkCategoryAntiquotParserFn (kind : Name) : ParserFn :=
16232   (mkCategoryAntiquotParser kind).fn
16233
16234 def categoryParserFnImpl (catName : Name) : ParserFn := fun ctx s =>
16235   let catName := if catName == `syntax then `stx else catName -- temporary Hack
16236   let categories := (parserExtension.getState ctx.env).categories
16237   match getCategory categories catName with
16238   | some cat =>
16239     prattParser catName cat.tables cat.behavior (mkCategoryAntiquotParserFn catName) ctx s
16240   | none      => s.mkUnexpectedError ("unknown parser category '" ++ toString catName ++ "'")
16241
16242 @[builtinInit] def setCategoryParserFnRef : IO Unit :=
16243   categoryParserFnRef.set categoryParserFnImpl
16244
16245 def addToken (tk : Token) (kind : AttributeKind) : AttrM Unit := do
16246   -- Recall that `ParserExtension.addEntry` is pure, and assumes `addTokenConfig` does not fail.
16247   -- So, we must run it here to handle exception.
16248   discard <| ofExcept <| addTokenConfig (parserExtension.getState (← getEnv)).tokens tk
16249   parserExtension.add (ParserExtension.Entry.token tk) kind
16250
16251 def addSyntaxNodeKind (env : Environment) (k : SyntaxNodeKind) : Environment :=
16252   parserExtension.addEntry env <| ParserExtension.Entry.kind k
16253
16254 def isValidSyntaxNodeKind (env : Environment) (k : SyntaxNodeKind) : Bool :=
16255   let kinds := (parserExtension.getState env).kinds
16256   kinds.contains k
16257
16258 def getSyntaxNodeKinds (env : Environment) : List SyntaxNodeKind := do
16259   let kinds := (parserExtension.getState env).kinds
16260   kinds.foldl (fun ks k _ => k::ks) []
16261

```

```

16262 def getTokenTable (env : Environment) : TokenTable :=
16263   (parserExtension.getState env).tokens
16264
16265 def mkInputContext (input : String) (fileName : String) : InputContext := {
16266   input      := input,
16267   fileName   := fileName,
16268   fileMap    := input.toFileMap
16269 }
16270
16271 def mkParserContext (ictx : InputContext) (pmctx : ParserModuleContext) : ParserContext := {
16272   prec                := 0,
16273   toInputContext      := ictx,
16274   toParserModuleContext := pmctx,
16275   tokens              := getTokenTable pmctx.env
16276 }
16277
16278 def mkParserState (input : String) : ParserState :=
16279   { cache := initCacheForInput input }
16280
16281 /- convenience function for testing -/
16282 def runParserCategory (env : Environment) (catName : Name) (input : String) (fileName := "<input>") : Except String Syntax :=
16283   let c := mkParserContext (mkInputContext input fileName) { env := env, options := {} }
16284   let s := mkParserState input
16285   let s := whitespace c s
16286   let s := categoryParserFnImpl catName c s
16287   if s.hasError then
16288     Except.error (s.toErrorMsg c)
16289   else if input.atEnd s.pos then
16290     Except.ok s.stxStack.back
16291   else
16292     Except.error ((s.mkError "end of input").toErrorMsg c)
16293
16294 def declareBuiltinParser (env : Environment) (addFnName : Name) (catName : Name) (declName : Name) (prio : Nat) : IO Environment :=
16295   let name := `__regBuiltinParser ++ declName
16296   let type := mkApp (mkConst `IO) (mkConst `Unit)
16297   let val  := mkAppN (mkConst addFnName) #[toExpr catName, toExpr declName, mkConst declName, mkNatLit prio]
16298   let decl := Declaration.defnDecl { name := name, levelParams := [], type := type, value := val, hints := ReducibilityHints.opaque,
16299                                     safety := DefinitionSafety.safe }
16300   match env.addAndCompile {} decl with
16301   -- TODO: pretty print error
16302   | Except.error _ => throw (IO.userError ("failed to emit registration code for builtin parser '" ++ toString declName ++ "'"))
16303   | Except.ok env  => IO.ofExcept (setBuiltinInitAttr env name)
16304
16305 def declareLeadingBuiltinParser (env : Environment) (catName : Name) (declName : Name) (prio : Nat) : IO Environment := -- TODO: use Co
16306   declareBuiltinParser env `Lean.Parser.addBuiltinLeadingParser catName declName prio
16307
16308 def declareTrailingBuiltinParser (env : Environment) (catName : Name) (declName : Name) (prio : Nat) : IO Environment := -- TODO: use C

```

```

16309 declareBuiltinParser env `Lean.Parser.addBuiltinTrailingParser catName declName prio
16310
16311 def getParserPriority (args : Syntax) : Except String Nat :=
16312   match args.getNumArgs with
16313   | 0 => pure 0
16314   | 1 => match (args.getArg 0).isNatLit? with
16315     | some prio => pure prio
16316     | none => throw "invalid parser attribute, numeral expected"
16317   | _ => throw "invalid parser attribute, no argument or numeral expected"
16318
16319 private def BuiltinParserAttribute.add (attrName : Name) (catName : Name)
16320   (declName : Name) (stx : Syntax) (kind : AttributeKind) : AttrM Unit := do
16321   let prio ← Attribute.Builtin.getPrio stx
16322   unless kind == AttributeKind.global do throwError "invalid attribute '{attrName}', must be global"
16323   let decl ← getConstInfo declName
16324   let env ← getEnv
16325   match decl.type with
16326   | Expr.const `Lean.Parser.TrailingParser _ _ => do
16327     let env ← declareTrailingBuiltinParser env catName declName prio
16328     setEnv env
16329   | Expr.const `Lean.Parser.Parser _ _ => do
16330     let env ← declareLeadingBuiltinParser env catName declName prio
16331     setEnv env
16332   | _ => throwError "unexpected parser type at '{declName}' (`Parser` or `TrailingParser` expected)"
16333   runParserAttributeHooks catName declName (builtin := true)
16334
16335 /-
16336 The parsing tables for builtin parsers are "stored" in the extracted source code.
16337 -/
16338 def registerBuiltinParserAttribute (attrName : Name) (catName : Name) (behavior := LeadingIdentBehavior.default) : IO Unit := do
16339   addBuiltinParserCategory catName behavior
16340   registerBuiltinAttribute {
16341     name      := attrName,
16342     descr     := "Builtin parser",
16343     add       := fun declName stx kind => liftM $ BuiltinParserAttribute.add attrName catName declName stx kind,
16344     applicationTime := AttributeApplicationTime.afterCompilation
16345   }
16346
16347 private def ParserAttribute.add (attrName : Name) (catName : Name) (declName : Name) (stx : Syntax) (attrKind : AttributeKind) : AttrM |
16348   let prio ← Attribute.Builtin.getPrio stx
16349   let env ← getEnv
16350   let opts ← getOptions
16351   let categories := (parserExtension.getState env).categories
16352   let p ← mkParserOfConstant categories declName
16353   let leading   := p.1
16354   let parser    := p.2
16355   let tokens    := parser.info.collectTokens []

```

```

16356 tokens.forM fun token => do
16357   try
16358     addToken token attrKind
16359   catch
16360     | Exception.error ref msg => throwError "invalid parser '{declName}', {msg}"
16361     | ex => throw ex
16362   let kinds := parser.info.collectKinds {}
16363   kinds.forM fun kind _ => modifyEnv fun env => addSyntaxNodeKind env kind
16364   let entry := ParserExtension.Entry.parser catName declName leading parser prio
16365   match addParser categories catName declName leading parser prio with
16366   | Except.error ex => throwError ex
16367   | Except.ok _      => parserExtension.add entry attrKind
16368   runParserAttributeHooks catName declName (builtin := false)
16369
16370 def mkParserAttributeImpl (attrName : Name) (catName : Name) : AttributeImpl where
16371   name                := attrName
16372   descr                := "parser"
16373   add declName stx attrKind := ParserAttribute.add attrName catName declName stx attrKind
16374   applicationTime        := AttributeApplicationTime.afterCompilation
16375
16376 /- A builtin parser attribute that can be extended by users. -/
16377 def registerBuiltinDynamicParserAttribute (attrName : Name) (catName : Name) : IO Unit := do
16378   registerBuiltinAttribute (mkParserAttributeImpl attrName catName)
16379
16380 @[builtinInit] private def registerParserAttributeImplBuilder : IO Unit :=
16381   registerAttributeImplBuilder `parserAttr fun args =>
16382     match args with
16383     | [DataValue.ofName attrName, DataValue.ofName catName] => pure $ mkParserAttributeImpl attrName catName
16384     | _ => throw "invalid parser attribute implementation builder arguments"
16385
16386 def registerParserCategory (env : Environment) (attrName : Name) (catName : Name) (behavior := LeadingIdentBehavior.default) : IO Envir
16387   let env ← IO.ofExcept $ addParserCategory env catName behavior
16388   registerAttributeOfBuilder env `parserAttr [DataValue.ofName attrName, DataValue.ofName catName]
16389
16390 -- declare `termParser` here since it is used everywhere via antiquotations
16391
16392 builtin_initialize registerBuiltinParserAttribute `builtinTermParser `term
16393
16394 builtin_initialize registerBuiltinDynamicParserAttribute `termParser `term
16395
16396 -- declare `commandParser` to break cyclic dependency
16397 builtin_initialize registerBuiltinParserAttribute `builtinCommandParser `command
16398
16399 builtin_initialize registerBuiltinDynamicParserAttribute `commandParser `command
16400
16401 @[inline] def commandParser (rbp : Nat := 0) : Parser :=
16402   categoryParser `command rbp

```

```

16403
16404 def notFollowedByCategoryTokenFn (catName : Name) : ParserFn := fun ctx s =>
16405   let categories := (parserExtension.getState ctx.env).categories
16406   match getCategory categories catName with
16407   | none => s.mkUnexpectedError s!"unknown parser category '{catName}'"
16408   | some cat =>
16409     let (s, stx) := peekToken ctx s
16410     match stx with
16411     | Except.ok (Syntax.atom _ sym) =>
16412       if ctx.insideQuot && sym == "$" then s
16413       else match cat.tables.leadingTable.find? (Name.mkSimple sym) with
16414       | some _ => s.mkUnexpectedError (toString catName)
16415       | _ => s
16416     | Except.ok _ => s
16417     | Except.error _ => s
16418
16419 @[inline] def notFollowedByCategoryToken (catName : Name) : Parser := {
16420   fn := notFollowedByCategoryTokenFn catName
16421 }
16422
16423 abbrev notFollowedByCommandToken : Parser :=
16424   notFollowedByCategoryToken `command
16425
16426 abbrev notFollowedByTermToken : Parser :=
16427   notFollowedByCategoryToken `term
16428
16429 end Parser
16430 end Lean
16431 ::::::::::::::
16432 Parser/Extra.lean
16433 ::::::::::::::
16434 /-
16435 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
16436 Released under Apache 2.0 license as described in the file LICENSE.
16437 Authors: Leonardo de Moura, Sebastian Ullrich
16438 -/
16439 import Lean.Parser.Extension
16440 -- necessary for auto-generation
16441 import Lean.PrettyPrinter.Parenthesizer
16442 import Lean.PrettyPrinter.Formatter
16443
16444 namespace Lean
16445 namespace Parser
16446
16447 -- synthesize pretty printers for parsers declared prior to `Lean.PrettyPrinter`
16448 -- (because `Parser.Extension` depends on them)
16449 attribute [runBuiltinParserAttributeHooks]

```

```

16450 leadingNode termParser commandParser mkAntiquot nodeWithAntiquot sepBy sepBy1
16451 unicodeSymbol nonReservedSymbol
16452
16453 @[runBuiltinParserAttributeHooks] def optional (p : Parser) : Parser :=
16454   optionalNoAntiquot (withAntiquotSpliceAndSuffix `optional p (symbol "?"))
16455
16456 @[runBuiltinParserAttributeHooks] def many (p : Parser) : Parser :=
16457   manyNoAntiquot (withAntiquotSpliceAndSuffix `many p (symbol "*"))
16458
16459 @[runBuiltinParserAttributeHooks] def many1 (p : Parser) : Parser :=
16460   many1NoAntiquot (withAntiquotSpliceAndSuffix `many p (symbol "*"))
16461
16462 @[runBuiltinParserAttributeHooks] def ident : Parser :=
16463   withAntiquot (mkAntiquot "ident" identKind) identNoAntiquot
16464
16465 -- `ident` and `rawIdent` produce the same syntax tree, so we reuse the antiquotation kind name
16466 @[runBuiltinParserAttributeHooks] def rawIdent : Parser :=
16467   withAntiquot (mkAntiquot "ident" identKind) rawIdentNoAntiquot
16468
16469 @[runBuiltinParserAttributeHooks] def numLit : Parser :=
16470   withAntiquot (mkAntiquot "numLit" numLitKind) numLitNoAntiquot
16471
16472 @[runBuiltinParserAttributeHooks] def scientificLit : Parser :=
16473   withAntiquot (mkAntiquot "scientificLit" scientificLitKind) scientificLitNoAntiquot
16474
16475 @[runBuiltinParserAttributeHooks] def strLit : Parser :=
16476   withAntiquot (mkAntiquot "strLit" strLitKind) strLitNoAntiquot
16477
16478 @[runBuiltinParserAttributeHooks] def charLit : Parser :=
16479   withAntiquot (mkAntiquot "charLit" charLitKind) charLitNoAntiquot
16480
16481 @[runBuiltinParserAttributeHooks] def nameLit : Parser :=
16482   withAntiquot (mkAntiquot "nameLit" nameLitKind) nameLitNoAntiquot
16483
16484 @[runBuiltinParserAttributeHooks, inline] def group (p : Parser) : Parser :=
16485   node groupKind p
16486
16487 @[runBuiltinParserAttributeHooks, inline] def many1Indent (p : Parser) : Parser :=
16488   withPosition $ many1 (checkColGe "irrelevant" >> p)
16489
16490 @[runBuiltinParserAttributeHooks, inline] def manyIndent (p : Parser) : Parser :=
16491   withPosition $ many (checkColGe "irrelevant" >> p)
16492
16493 @[runBuiltinParserAttributeHooks] def abbrev notSymbol (s : String) : Parser :=
16494   notFollowedBy (symbol s) s
16495
16496 /-- No-op parser that advises the pretty printer to emit a non-breaking space. -/

```

```

16497 @[inline] def ppHardSpace : Parser := skip
16498 /-- No-op parser that advises the pretty printer to emit a space/soft line break. -/
16499 @[inline] def ppSpace : Parser := skip
16500 /-- No-op parser that advises the pretty printer to emit a hard line break. -/
16501 @[inline] def ppLine : Parser := skip
16502 /--
16503   No-op parser combinator that advises the pretty printer to group and indent the given syntax.
16504   By default, only syntax categories are grouped. -/
16505 @[inline] def ppGroup : Parser → Parser := id
16506 /-- No-op parser combinator that advises the pretty printer to indent the given syntax without grouping it. -/
16507 @[inline] def ppIndent : Parser → Parser := id
16508 /--
16509   No-op parser combinator that advises the pretty printer to dedent the given syntax.
16510   Dedenting can in particular be used to counteract automatic indentation. -/
16511 @[inline] def ppDedent : Parser → Parser := id
16512
16513 end Parser
16514
16515 section
16516 open PrettyPrinter
16517
16518 @[combinatorFormatter Lean.Parser.ppHardSpace] def ppHardSpace.formatter : Formatter := Formatter.push " "
16519 @[combinatorFormatter Lean.Parser.ppSpace] def ppSpace.formatter : Formatter := Formatter.pushLine
16520 @[combinatorFormatter Lean.Parser.ppLine] def ppLine.formatter : Formatter := Formatter.push "\n"
16521 @[combinatorFormatter Lean.Parser.ppGroup] def ppGroup.formatter (p : Formatter) : Formatter := Formatter.group $ Formatter.indent p
16522 @[combinatorFormatter Lean.Parser.ppIndent] def ppIndent.formatter (p : Formatter) : Formatter := Formatter.indent p
16523 @[combinatorFormatter Lean.Parser.ppDedent] def ppDedent.formatter (p : Formatter) : Formatter := do
16524   let opts ← getOptions
16525   Formatter.indent p (some ((0:Int) - Std.Format.getIndent opts))
16526 end
16527
16528 namespace Parser
16529
16530 -- now synthesize parenthesizers
16531 attribute [runBuiltinParserAttributeHooks]
16532 ppHardSpace ppSpace ppLine ppGroup ppIndent ppDedent
16533
16534 macro "register_parser_alias" aliasName:strLit declName:ident : term =>
16535   `(do Parser.registerAlias $aliasName $declName
16536     PrettyPrinter.Formatter.registerAlias $aliasName $(mkIdentFrom declName (declName.getId ++ `formatter))
16537     PrettyPrinter.Parenthesizer.registerAlias $aliasName $(mkIdentFrom declName (declName.getId ++ `parenthesizer)))
16538
16539 builtin_initialize
16540   register_parser_alias "group" group
16541   register_parser_alias "ppHardSpace" ppHardSpace
16542   register_parser_alias "ppSpace" ppSpace
16543   register_parser_alias "ppLine" ppLine

```



```

16544   register_parser_alias "ppGroup" ppGroup
16545   register_parser_alias "ppIndent" ppIndent
16546   register_parser_alias "ppDedent" ppDedent
16547
16548 end Parser
16549
16550 open Parser
16551
16552 open PrettyPrinter.Parenthesizer (registerAlias) in
16553 builtin_initialize
16554   registerAlias "num" numLit.parenthesizer
16555   registerAlias "scientific" scientificLit.parenthesizer
16556   registerAlias "str" strLit.parenthesizer
16557   registerAlias "char" charLit.parenthesizer
16558   registerAlias "name" nameLit.parenthesizer
16559   registerAlias "ident" ident.parenthesizer
16560   registerAlias "many" many.parenthesizer
16561   registerAlias "many1" many1.parenthesizer
16562   registerAlias "optional" optional.parenthesizer
16563
16564 open PrettyPrinter.Formatter (registerAlias) in
16565 builtin_initialize
16566   registerAlias "num" numLit.formatter
16567   registerAlias "scientific" scientificLit.formatter
16568   registerAlias "str" strLit.formatter
16569   registerAlias "char" charLit.formatter
16570   registerAlias "name" nameLit.formatter
16571   registerAlias "ident" ident.formatter
16572   registerAlias "many" many.formatter
16573   registerAlias "many1" many1.formatter
16574   registerAlias "optional" optional.formatter
16575
16576 end Lean
16577 ::::::::::::::
16578 Parser/Level.lean
16579 ::::::::::::::
16580 /-
16581 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
16582 Released under Apache 2.0 license as described in the file LICENSE.
16583 Authors: Leonardo de Moura, Sebastian Ullrich
16584 -/
16585 import Lean.Parser.Extra
16586
16587 namespace Lean
16588 namespace Parser
16589
16590 builtin_initialize

```

```

16591 registerBuiltinParserAttribute `builtinLevelParser `level
16592
16593 @[inline] def levelParser (rbp : Nat := 0) : Parser :=
16594   categoryParser `level rbp
16595
16596 namespace Level
16597
16598 @[builtinLevelParser] def paren := leading_parser "(" >> levelParser >> ")"
16599 @[builtinLevelParser] def max  := leading_parser nonReservedSymbol "max" true >> many1 (ppSpace >> levelParser maxPrec)
16600 @[builtinLevelParser] def imax := leading_parser nonReservedSymbol "imax" true >> many1 (ppSpace >> levelParser maxPrec)
16601 @[builtinLevelParser] def hole := leading_parser "_"
16602 @[builtinLevelParser] def num  := checkPrec maxPrec >> numLit
16603 @[builtinLevelParser] def ident := checkPrec maxPrec >> Parser.ident
16604 @[builtinLevelParser] def addLit := trailing_parser:65 " + " >> numLit
16605
16606 end Level
16607
16608 end Parser
16609 end Lean
16610 ::::::::::::::
16611 Parser/Module.lean
16612 ::::::::::::::
16613 /-
16614 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
16615 Released under Apache 2.0 license as described in the file LICENSE.
16616 Authors: Leonardo de Moura, Sebastian Ullrich
16617 -/
16618 import Lean.Message
16619 import Lean.Parser.Command
16620
16621 namespace Lean
16622 namespace Parser
16623
16624 namespace Module
16625 def «prelude» := leading_parser "prelude"
16626 def «import»  := leading_parser "import " >> optional "runtime" >> ident
16627 def header    := leading_parser optional («prelude» >> ppline) >> many («import» >> ppline) >> ppline
16628 /--
16629   Parser for a Lean module. We never actually run this parser but instead use the imperative definitions below that
16630   return the same syntax tree structure, but add error recovery. Still, it is helpful to have a `Parser` definition
16631   for it in order to auto-generate helpers such as the pretty printer. -/
16632 @[runBuiltinParserAttributeHooks]
16633 def module    := leading_parser header >> many (commandParser >> ppline >> ppline)
16634
16635 def updateTokens (c : ParserContext) : ParserContext :=
16636   { c with
16637     tokens := match addParserTokens c.tokens header.info with

```

```

16638     | Except.ok tables => tables
16639     | Except.error _   => unreachable! }
16640
16641 end Module
16642
16643 structure ModuleParserState where
16644   pos      : String.Pos := 0
16645   recovering : Bool      := false
16646   deriving Inhabited
16647
16648 private def mkErrorMessage (c : ParserContext) (pos : String.Pos) (errorMsg : String) : Message :=
16649   let pos := c.fileMap.toPosition pos
16650   { fileName := c.fileName, pos := pos, data := errorMsg }
16651
16652 def parseHeader (inputCtx : InputContext) : IO (Syntax × ModuleParserState × MessageLog) := do
16653   let dummyEnv ← mkEmptyEnvironment
16654   let ctx := mkParserContext inputCtx { env := dummyEnv, options := {} }
16655   let ctx := Module.updateTokens ctx
16656   let s   := mkParserState ctx.input
16657   let s   := whitespace ctx s
16658   let s   := Module.header.fn ctx s
16659   let stx := s.stxStack.back
16660   match s.errorMsg with
16661   | some errorMsg =>
16662     let msg := mkErrorMessage ctx s.pos (toString errorMsg)
16663     pure (stx, { pos := s.pos, recovering := true }, { : MessageLog }.add msg)
16664   | none =>
16665     pure (stx, { pos := s.pos }, {})
16666
16667 private def mkEOI (pos : String.Pos) : Syntax :=
16668   let atom := mkAtom (SourceInfo.original "".toSubstring pos "".toSubstring) ""
16669   Syntax.node `Lean.Parser.Module.eoi #[atom]
16670
16671 def isEOI (s : Syntax) : Bool :=
16672   s.isOfKind `Lean.Parser.Module.eoi
16673
16674 def isExitCommand (s : Syntax) : Bool :=
16675   s.isOfKind `Lean.Parser.Command.exit
16676
16677 private def consumeInput (c : ParserContext) (pos : String.Pos) : String.Pos :=
16678   let s : ParserState := { cache := initCacheForInput c.input, pos := pos }
16679   let s := tokenFn [] c s
16680   match s.errorMsg with
16681   | some _ => pos + 1
16682   | none   => s.pos
16683
16684 def topLevelCommandParserFn : ParserFn :=

```

```

16685 commandParser.fn
16686
16687 partial def parseCommand (inputCtx : InputContext) (pmctx : ParserModuleContext) (s : ModuleParserState) (messages : MessageLog) : Synt.
16688 let rec parse (s : ModuleParserState) (messages : MessageLog) :=
16689   let { pos := pos, recovering := recovering } := s
16690   if inputCtx.input.atEnd pos then
16691     (mkEOI pos, s, messages)
16692   else
16693     let c := mkParserContext inputCtx pmctx
16694     let s := { cache := initCacheForInput c.input, pos := pos : ParserState }
16695     let s := whitespace c s
16696     let s := topLevelCommandParserFn c s
16697     match s.errorMsg with
16698     | none =>
16699       (s.stxStack.back, { pos := s.pos }, messages)
16700     | some errorMsg =>
16701       -- advance at least one token to prevent infinite loops
16702       let pos := if s.pos == pos then consumeInput c s.pos else s.pos
16703       /- We ignore commands where `getPos?` is none. This happens only on commands that have a prefix comprised of optional elements.
16704          For example, unification hints start with `optional («scoped» <|> «local»)` .
16705          We claim a syntactically incorrect command containing no token or identifier is irrelevant for intellisense and should be ig.
16706       let ignore := s.stxStack.isEmpty || s.stxStack.back.getPos?.isNone
16707       let messages := if recovering && ignore then messages else messages.add <| mkErrorMessage c s.pos (toString errorMsg)
16708       if ignore then
16709         parse { pos := pos, recovering := true } messages
16710       else
16711         (s.stxStack.back, { pos := pos, recovering := true }, messages)
16712 parse s messages
16713
16714 -- only useful for testing since most Lean files cannot be parsed without elaboration
16715
16716 partial def testParseModuleAux (env : Environment) (inputCtx : InputContext) (s : ModuleParserState) (msgs : MessageLog) (stxs : Array
16717 let rec parse (state : ModuleParserState) (msgs : MessageLog) (stxs : Array Syntax) :=
16718   match parseCommand inputCtx { env := env, options := {} } state msgs with
16719   | (stx, state, msgs) =>
16720     if isEOI stx then
16721       if msgs.isEmpty then
16722         pure stxs
16723       else do
16724         msgs.forM fun msg => msg.toString >=> IO.println
16725         throw (IO.userError "failed to parse file")
16726     else
16727       parse state msgs (stxs.push stx)
16728 parse s msgs stxs
16729
16730 def testParseModule (env : Environment) (fname contents : String) : IO Syntax := do
16731   let fname ← IO.realPath fname

```

```

16732 let inputCtx := mkInputContext contents fname
16733 let (header, state, messages) ← parseHeader inputCtx
16734 let cmds ← testParseModuleAux env inputCtx state messages #[]
16735 let stx := Syntax.node `Lean.Parser.Module.module #[header, mkListNode cmds]
16736 pure stx.updateLeading
16737
16738 def testParseFile (env : Environment) (fname : String) : IO Syntax := do
16739   let contents ← IO.FS.readFile fname
16740   testParseModule env fname contents
16741
16742 end Parser
16743 end Lean
16744 ::::::::::::::
16745 Parser/StrInterpolation.lean
16746 ::::::::::::::
16747 /-
16748 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
16749 Released under Apache 2.0 license as described in the file LICENSE.
16750 Authors: Leonardo de Moura
16751 -/
16752 import Lean.Parser.Basic
16753 namespace Lean.Parser
16754
16755 def isQuotableCharForStrInterpolant (c : Char) : Bool :=
16756   c == '{' || isQuotableCharDefault c
16757
16758 partial def interpolatedStrFn (p : ParserFn) : ParserFn := fun c s =>
16759   let input      := c.input
16760   let stackSize := s.stackSize
16761   let rec parse (startPos : Nat) (c : ParserContext) (s : ParserState) : ParserState :=
16762     let i      := s.pos
16763     if input.atEnd i then
16764       s.mkUnexpectedErrorAt "unterminated string literal" startPos
16765     else
16766       let curr := input.get i
16767       let s     := s.setPos (input.next i)
16768       if curr == '\"' then
16769         let s := mkNodeToken interpolatedStrLitKind startPos c s
16770         s.mkNode interpolatedStrKind stackSize
16771       else if curr == '\\\' then
16772         andthenFn (quotedCharCoreFn isQuotableCharForStrInterpolant) (parse startPos) c s
16773       else if curr == '{' then
16774         let s := mkNodeToken interpolatedStrLitKind startPos c s
16775         let s := p c s
16776         if s.hasError then s
16777       else
16778         let pos := s.pos

```

```

16779         andthenFn (satisfyFn (· == '}') "expected '}'") (parse pos) c s
16780     else
16781         parse startPos c s
16782     let startPos := s.pos
16783     if input.atEnd startPos then
16784         s.mkEOLError
16785     else
16786         let curr := input.get s.pos;
16787         if curr != '\"' then
16788             s.mkError "interpolated string"
16789         else
16790             let s := s.next input startPos
16791             parse startPos c s
16792
16793 @[inline] def interpolatedStrNoAntiquot (p : Parser) : Parser := {
16794     fn := interpolatedStrFn p.fn,
16795     info := mkAtomicInfo "interpolatedStr"
16796 }
16797
16798 def interpolatedStr (p : Parser) : Parser :=
16799     withAntiquot (mkAntiquot "interpolatedStr" interpolatedStrKind) $ interpolatedStrNoAntiquot p
16800
16801 end Lean.Parser
16802 ::::::::::::::
16803 Parser/Syntax.lean
16804 ::::::::::::::
16805 /-
16806 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
16807 Released under Apache 2.0 license as described in the file LICENSE.
16808 Authors: Leonardo de Moura, Sebastian Ullrich
16809 -/
16810 import Lean.Parser.Command
16811 import Lean.Parser.Tactic
16812
16813 namespace Lean
16814 namespace Parser
16815
16816 builtin_initialize
16817     registerBuiltinParserAttribute `builtinSyntaxParser `stx LeadingIdentBehavior.both
16818     registerBuiltinDynamicParserAttribute `stxParser `stx
16819
16820 builtin_initialize
16821     registerBuiltinParserAttribute `builtinPrecParser `prec LeadingIdentBehavior.both
16822     registerBuiltinDynamicParserAttribute `precParser `prec
16823
16824 @[inline] def precedenceParser (rbp : Nat := 0) : Parser :=
16825     categoryParser `prec rbp

```

```

16826
16827 @[inline] def syntaxParser (rbp : Nat := 0) : Parser :=
16828   categoryParser `stx rbp
16829
16830 def «precedence» := leading_parser ":" >> precedenceParser maxPrec
16831 def optPrecedence := optional (atomic «precedence»)
16832
16833 namespace Syntax
16834 @[builtinPrecParser] def numPrec := checkPrec maxPrec >> numLit
16835
16836 @[builtinSyntaxParser] def paren      := leading_parser "(" >> many1 syntaxParser >> ")"
16837 @[builtinSyntaxParser] def cat       := leading_parser ident >> optPrecedence
16838 @[builtinSyntaxParser] def unary    := leading_parser ident >> checkNoWsBefore >> "(" >> many1 syntaxParser >> ")"
16839 @[builtinSyntaxParser] def binary   := leading_parser ident >> checkNoWsBefore >> "(" >> many1 syntaxParser >> ", " >> many1 syn
16840 @[builtinSyntaxParser] def sepBy    := leading_parser "sepBy(" >> many1 syntaxParser >> ", " >> strLit >> optional (" " >> many
16841 @[builtinSyntaxParser] def sepBy1   := leading_parser "sepBy1(" >> many1 syntaxParser >> ", " >> strLit >> optional (" " >> man
16842 @[builtinSyntaxParser] def atom     := leading_parser strLit
16843 @[builtinSyntaxParser] def nonReserved := leading_parser "&" >> strLit
16844
16845 end Syntax
16846
16847 namespace Term
16848
16849 @[builtinTermParser] def stx.quot : Parser := leading_parser "`(stx|" >> toggleInsideQuot syntaxParser >> ")"
16850 @[builtinTermParser] def prec.quot : Parser := leading_parser "`(prec|" >> toggleInsideQuot precedenceParser >> ")"
16851 @[builtinTermParser] def prio.quot : Parser := leading_parser "`(prio|" >> toggleInsideQuot priorityParser >> ")"
16852
16853 end Term
16854
16855 namespace Command
16856
16857 def namedName := leading_parser (atomic "(" >> nonReservedSymbol "name") >> " := " >> ident >> ")")
16858 def optNamedName := optional namedName
16859
16860 def «prefix» := leading_parser "prefix"
16861 def «infix» := leading_parser "infix"
16862 def «infixl» := leading_parser "infixl"
16863 def «infixr» := leading_parser "infixr"
16864 def «postfix» := leading_parser "postfix"
16865 def mixfixKind := «prefix» <|> «infix» <|> «infixl» <|> «infixr» <|> «postfix»
16866 @[builtinCommandParser] def «mixfix» := leading_parser Term.attrKind >> mixfixKind >> optPrecedence >> optNamedName >> optNamedPrio >
16867 -- NOTE: We use `suppressInsideQuot` in the following parsers because quotations inside them are evaluated in the same stage and
16868 -- thus should be ignored when we use `checkInsideQuot` to prepare the next stage for a builtin syntax change
16869 def identPrec := leading_parser ident >> optPrecedence
16870
16871 def optKind : Parser := optional "(" >> nonReservedSymbol "kind" >> " := " >> ident >> ")")
16872

```

```

16873 def notationItem := ppSpace >> withAntiquot (mkAntiquot "notationItem" `Lean.Parser.Command.notationItem) (strLit <|> identPrec)
16874 @[builtinCommandParser] def «notation» := leading_parser Term.attrKind >> "notation" >> optPrecedence >> optNamedName >> optNamedPri
16875 @[builtinCommandParser] def «macro_rules» := suppressInsideQuot (leading_parser "macro_rules" >> optKind >> Term.matchAlts)
16876 @[builtinCommandParser] def «syntax» := leading_parser Term.attrKind >> "syntax" >> optPrecedence >> optNamedName >> optNamedPrio
16877 @[builtinCommandParser] def syntaxAbbrev := leading_parser "syntax" >> ident >> " := " >> many1 syntaxParser
16878 @[builtinCommandParser] def syntaxCat := leading_parser "declare_syntax_cat" >> ident
16879 def macroArgSimple := leading_parser ident >> checkNoWsBefore "no space before ':'" >> ":" >> syntaxParser maxPrec
16880 def macroArgSymbol := leading_parser strLit >> optional (atomic <| checkNoWsBefore >> "%" >> checkNoWsBefore >> ident)
16881 def macroArg := macroArgSymbol <|> atomic macroArgSimple
16882 def macroHead := macroArg
16883 def macroTailTactic : Parser := atomic (" : " >> identEq "tactic") >> darrow >> ("`(" >> toggleInsideQuot Tactic.seq1 >> ")") <|> term
16884 def macroTailCommand : Parser := atomic (" : " >> identEq "command") >> darrow >> ("`(" >> toggleInsideQuot (many1Unbox commandParser)
16885 def macroTailDefault : Parser := atomic (" : " >> ident) >> darrow >> ("`(" >> toggleInsideQuot (categoryParserOfStack 2) >> ")") <|>
16886 def macroTail := macroTailTactic <|> macroTailCommand <|> macroTailDefault
16887 @[builtinCommandParser] def «macro» := leading_parser suppressInsideQuot (Term.attrKind >> "macro" >> optPrecedence >> optNamedN
16888
16889 @[builtinCommandParser] def «elab_rules» := leading_parser suppressInsideQuot ("elab_rules" >> optKind >> optional (" : " >> ident) >> `
16890 def elabHead := macroHead
16891 def elabArg := macroArg
16892 def elabTail := atomic (" : " >> ident >> optional (" <= " >> ident)) >> darrow >> termParser
16893 @[builtinCommandParser] def «elab» := leading_parser suppressInsideQuot (Term.attrKind >> "elab" >> optPrecedence >> optNamedNam
16894
16895 end Command
16896
16897 end Parser
16898 end Lean
16899 ::::::::::::::
16900 Parser/Tactic.lean
16901 ::::::::::::::
16902 /-
16903 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
16904 Released under Apache 2.0 license as described in the file LICENSE.
16905 Authors: Leonardo de Moura, Sebastian Ullrich
16906 -/
16907 import Lean.Parser.Term
16908
16909 namespace Lean
16910 namespace Parser
16911 namespace Tactic
16912
16913 builtin_initialize
16914   register_parser_alias "tacticSeq" tacticSeq
16915
16916 @[builtinTacticParser] def «unknown» := leading_parser withPosition (ident >> errorAtSavedPos "unknown tactic" true)
16917 @[builtinTacticParser] def nestedTactic := tacticSeqBracketed
16918
16919 /- Auxiliary parser for expanding `match` tactic -/

```



```

16920 @[builtinTacticParser] def eraseAuxDiscrs := leading_parser:maxPrec "eraseAuxDiscrs!"
16921
16922 def matchRhs := Term.hole <|> Term.syntheticHole <|> tacticSeq
16923 def matchAlts := Term.matchAlts (rhsParser := matchRhs)
16924 @[builtinTacticParser] def «match» := leading_parser:leadPrec "match " >> optional Term.generalizingParam >> sepBy1 Term.matchDiscr ",
16925 @[builtinTacticParser] def introMatch := leading_parser nonReservedSymbol "intro " >> matchAlts
16926
16927 @[builtinTacticParser] def decide := leading_parser nonReservedSymbol "decide"
16928 @[builtinTacticParser] def nativeDecide := leading_parser nonReservedSymbol "nativeDecide"
16929
16930 end Tactic
16931 end Parser
16932 end Lean
16933 ::::::::::::::
16934 Parser/Term.lean
16935 ::::::::::::::
16936 /-
16937 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
16938 Released under Apache 2.0 license as described in the file LICENSE.
16939 Authors: Leonardo de Moura, Sebastian Ullrich
16940 -/
16941 import Lean.Parser.Attr
16942 import Lean.Parser.Level
16943
16944 namespace Lean
16945 namespace Parser
16946
16947 namespace Command
16948 def commentBody : Parser :=
16949 { fn := rawFn (fun c s => finishCommentBlock s.pos 1 c s) (trailingWs := true) }
16950
16951 @[combinatorParenthesizer Lean.Parser.Command.commentBody] def commentBody.parenthesizer := PrettyPrinter.Parenthesizer.visitToken
16952 @[combinatorFormatter Lean.Parser.Command.commentBody] def commentBody.formatter := PrettyPrinter.Formatter.visitAtom Name.anonymous
16953
16954 def docComment      := leading_parser ppDedent $ "/--" >> commentBody >> ppLine
16955 end Command
16956
16957 builtin_initialize
16958   registerBuiltinParserAttribute `builtinTacticParser `tactic LeadingIdentBehavior.both
16959   registerBuiltinDynamicParserAttribute `tacticParser `tactic
16960
16961 @[inline] def tacticParser (rbp : Nat := 0) : Parser :=
16962   categoryParser `tactic rbp
16963
16964 namespace Tactic
16965
16966 def tacticSeq1Indented : Parser :=

```

```

16967 leading_parser many1Indent (group (ppLine >> tacticParser >> optional ";"))
16968 def tacticSeqBracketed : Parser :=
16969   leading_parser "{" >> many (group (ppLine >> tacticParser >> optional ";")) >> ppDedent (ppLine >> "}")
16970 def tacticSeq :=
16971   nodeWithAntiquot "tacticSeq" `Lean.Parser.Tactic.tacticSeq (tacticSeqBracketed <|> tacticSeq1Indented)
16972
16973 /- Raw sequence for quotation and grouping -/
16974 def seq1 :=
16975   node `Lean.Parser.Tactic.seq1 $ sepBy1 tacticParser ";\n" (allowTrailingSep := true)
16976
16977 end Tactic
16978
16979 def darrow : Parser := " => "
16980
16981 namespace Term
16982
16983 /- Built-in parsers -/
16984
16985 @[builtinTermParser] def byTactic := leading_parser:leadPrec "by " >> Tactic.tacticSeq
16986
16987 def optSemicolon (p : Parser) : Parser := ppDedent $ optional ";" >> ppLine >> p
16988
16989 -- `checkPrec` necessary for the pretty printer
16990 @[builtinTermParser] def ident := checkPrec maxPrec >> Parser.ident
16991 @[builtinTermParser] def num : Parser := checkPrec maxPrec >> numLit
16992 @[builtinTermParser] def scientific : Parser := checkPrec maxPrec >> scientificLit
16993 @[builtinTermParser] def str : Parser := checkPrec maxPrec >> strLit
16994 @[builtinTermParser] def char : Parser := checkPrec maxPrec >> charLit
16995 @[builtinTermParser] def type := leading_parser "Type" >> optional (checkWsBefore "" >> checkPrec leadPrec >> checkColGt >> levelParser
16996 @[builtinTermParser] def sort := leading_parser "Sort" >> optional (checkWsBefore "" >> checkPrec leadPrec >> checkColGt >> levelParser
16997 @[builtinTermParser] def prop := leading_parser "Prop"
16998 @[builtinTermParser] def hole := leading_parser "_"
16999 @[builtinTermParser] def syntheticHole := leading_parser "?" >> (ident <|> hole)
17000 @[builtinTermParser] def «sorry» := leading_parser "sorry"
17001 @[builtinTermParser] def cdot := leading_parser symbol "." <|> "."
17002 @[builtinTermParser] def emptyC := leading_parser "∅" <|> (symbol "{" >> "}")
17003 def typeAscription := leading_parser " : " >> termParser
17004 def tupleTail := leading_parser ", " >> sepBy1 termParser ", "
17005 def parenSpecial : Parser := optional (tupleTail <|> typeAscription)
17006 @[builtinTermParser] def paren := leading_parser "(" >> ppDedent (withoutPosition (withoutForbidden (optional (termParser >> parenSpeci.
17007 @[builtinTermParser] def anonymousCtor := leading_parser "(" >> sepBy termParser ", " >> ")")
17008 def optIdent : Parser := optional (atomic (ident >> " : "))
17009 def fromTerm := leading_parser " from " >> termParser
17010 def haveAssign := leading_parser " := " >> termParser
17011 def haveDecl := leading_parser optIdent >> termParser >> (haveAssign <|> fromTerm <|> byTactic)
17012 @[builtinTermParser] def «have» := leading_parser:leadPrec withPosition ("have " >> haveDecl) >> optSemicolon termParser
17013 def sufficesDecl := leading_parser optIdent >> termParser >> (fromTerm <|> byTactic)

```

```

17014 @[builtinTermParser] def «suffices» := leading_parser:leadPrec withPosition ("suffices " >> sufficesDecl) >> optSemicolon termParser
17015 @[builtinTermParser] def «show»      := leading_parser:leadPrec "show " >> termParser >> (fromTerm <|> byTactic)
17016 def structInstArrayRef := leading_parser "[" >> termParser >> "]"
17017 def structInstLVal     := leading_parser (ident <|> fieldIdx <|> structInstArrayRef) >> many (group ( "." >> (ident <|> fieldIdx)) <|> str
17018 def structInstField    := ppGroup $ leading_parser structInstLVal >> " := " >> termParser
17019 def optEllipsis        := leading_parser optional ".."
17020 @[builtinTermParser] def structInst := leading_parser "{" >> ppHardSpace >> optional (atomic (termParser >> " with "))
17021   >> manyIndent (group (structInstField >> optional ", "))
17022   >> optEllipsis
17023   >> optional (" : " >> termParser) >> " }"
17024 def typeSpec := leading_parser " : " >> termParser
17025 def optType : Parser := optional typeSpec
17026 @[builtinTermParser] def explicit := leading_parser "@" >> termParser maxPrec
17027 @[builtinTermParser] def inaccessible := leading_parser "." >> termParser >> ")"
17028 def binderIdent : Parser := ident <|> hole
17029 def binderType (requireType := false) : Parser := if requireType then node nullKind (" : " >> termParser) else optional (" : " >> termP
17030 def binderTactic := leading_parser atomic (symbol " := " >> " by ") >> Tactic.tacticSeq
17031 def binderDefault := leading_parser " := " >> termParser
17032 def explicitBinder (requireType := false) := ppGroup $ leading_parser "(" >> many1 binderIdent >> binderType requireType >> optional (b
17033 def implicitBinder (requireType := false) := ppGroup $ leading_parser "{" >> many1 binderIdent >> binderType requireType >> "}"
17034 def instBinder := ppGroup $ leading_parser "[" >> optIdent >> termParser >> "]"
17035 def bracketedBinder (requireType := false) := withAntiquot (mkAntiquot "bracketedBinder" none (anonymous := false)) <|
17036   explicitBinder requireType <|> implicitBinder requireType <|> instBinder
17037
17038 /-
17039 It is feasible to support dependent arrows such as  $\{\alpha\} \rightarrow \alpha \rightarrow \alpha$  without sacrificing the quality of the error messages for the longer c
17040  $\{\alpha\} \rightarrow \alpha \rightarrow \alpha$  would be short for  $\{\alpha : \text{Type}\} \rightarrow \alpha \rightarrow \alpha$ 
17041 Here is the encoding:
17042 ```
17043 def implicitShortBinder := node `Lean.Parser.Term.implicitBinder $ "{" >> many1 binderIdent >> pushNone >> "}"
17044 def depArrowShortPrefix := try (implicitShortBinder >> unicodeSymbol " → " " -> ")
17045 def depArrowLongPrefix  := bracketedBinder true >> unicodeSymbol " → " " -> "
17046 def depArrowPrefix      := depArrowShortPrefix <|> depArrowLongPrefix
17047 @[builtinTermParser] def depArrow := leading_parser depArrowPrefix >> termParser
17048 ```
17049 Note that no changes in the elaborator are needed.
17050 We decided to not use it because terms such as  $\{\alpha\} \rightarrow \alpha \rightarrow \alpha$  may look too cryptic.
17051 Note that we did not add a `explicitShortBinder` parser since  $(\alpha) \rightarrow \alpha \rightarrow \alpha$  is really cryptic as a short for  $(\alpha : \text{Type}) \rightarrow \alpha \rightarrow \alpha$ .
17052 -/
17053 @[builtinTermParser] def depArrow := leading_parser:25 bracketedBinder true >> unicodeSymbol " → " " -> " >> termParser
17054
17055 def simpleBinder := leading_parser many1 binderIdent >> optType
17056 @[builtinTermParser]
17057 def «forall» := leading_parser:leadPrec unicodeSymbol "∀" "forall" >> many1 (ppSpace >> (simpleBinder <|> bracketedBinder)) >> ", " >>
17058
17059 def matchAlt (rhsParser : Parser := termParser) : Parser :=
17060   nodeWithAntiquot "matchAlt" `Lean.Parser.Term.matchAlt $

```

```

17061      "| " >> ppIndent (sepBy1 termParser ", " >> darrow >> checkColGe "alternative right-hand-side to start in a column greater than or
17062 /--
17063 Useful for syntax quotations. Note that generic patterns such as `` `(matchAltExpr| | ... => $rhs) `` should also
17064 work with other `rhsParser`s (of arity 1). -/
17065 def matchAltExpr := matchAlt
17066
17067 def matchAlts (rhsParser : Parser := termParser) : Parser :=
17068   leading_parser ppDedent $ withPosition $ many1Indent (ppLine >> matchAlt rhsParser)
17069
17070 def matchDiscr := leading_parser optional (atomic (ident >> checkNoWsBefore "no space before ':'" >> ":")) >> termParser
17071
17072 def trueVal := leading_parser nonReservedSymbol "true"
17073 def falseVal := leading_parser nonReservedSymbol "false"
17074 def generalizingParam := leading_parser atomic ("(" >> nonReservedSymbol "generalizing") >> " := " >> (trueVal <|> falseVal) >> ")")
17075
17076 @[builtinTermParser] def «match» := leading_parser:leadPrec "match " >> optional generalizingParam >> sepBy1 matchDiscr ", " >> optType
17077 @[builtinTermParser] def «nomatch» := leading_parser:leadPrec "nomatch " >> termParser
17078
17079 def funImplicitBinder := atomic (lookahead ("{" >> many1 binderIdent >> (symbol " : " <|> "}"))) >> implicitBinder
17080 def funSimpleBinder := atomic (lookahead (many1 binderIdent >> " : ")) >> simpleBinder
17081 def funBinder : Parser := funImplicitBinder <|> instBinder <|> funSimpleBinder <|> termParser maxPrec
17082 -- NOTE: we use `nodeWithAntiquot` to ensure that `fun $b => ...` remains a `term` antiquotation
17083 def basicFun : Parser := nodeWithAntiquot "basicFun" `Lean.Parser.Term.basicFun (many1 (ppSpace >> funBinder) >> darrow >> termParser)
17084 @[builtinTermParser] def «fun» := leading_parser:maxPrec unicodeSymbol "λ" "fun" >> (basicFun <|> matchAlts)
17085
17086 def optExprPrecedence := optional (atomic ":" >> termParser maxPrec)
17087 @[builtinTermParser] def «leading_parser» := leading_parser:leadPrec "leading_parser " >> optExprPrecedence >> termParser
17088 @[builtinTermParser] def «trailing_parser» := leading_parser:leadPrec "trailing_parser " >> optExprPrecedence >> optExprPrecedence >> t
17089
17090 @[builtinTermParser] def borrowed := leading_parser "@&" >> termParser leadPrec
17091 @[builtinTermParser] def quotedName := leading_parser nameLit
17092 @[builtinTermParser] def doubleQuotedName := leading_parser "\"" >> checkNoWsBefore >> nameLit
17093
17094 def simpleBinderWithoutType := nodeWithAntiquot "simpleBinder" `Lean.Parser.Term.simpleBinder (anonymous := true)
17095   (many1 binderIdent >> pushNone)
17096
17097 /- 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`. -/
17098 def letIdLhs : Parser := ident >> checkWsBefore "expected space before binders" >> many (ppSpace >> (simpleBinderWithoutType <|> bra
17099 def letIdDecl := nodeWithAntiquot "letIdDecl" `Lean.Parser.Term.letIdDecl $ atomic (letIdLhs >> " := ") >> termParser
17100 def letPatDecl := nodeWithAntiquot "letPatDecl" `Lean.Parser.Term.letPatDecl $ atomic (termParser >> pushNone >> optType >> " := ") :
17101 def letEqnsDecl := nodeWithAntiquot "letEqnsDecl" `Lean.Parser.Term.letEqnsDecl $ letIdLhs >> matchAlts
17102 -- Remark: we use `nodeWithAntiquot` here to make sure anonymous antiquotations (e.g., `$x`) are not `letDecl`
17103 def letDecl := nodeWithAntiquot "letDecl" `Lean.Parser.Term.letDecl (notFollowedBy (nonReservedSymbol "rec") "rec" >> (letIdDecl <|>
17104 @[builtinTermParser] def «let» := leading_parser:leadPrec withPosition ("let " >> letDecl) >> optSemicolon termParser
17105 @[builtinTermParser] def «let_fun» := leading_parser:leadPrec withPosition ((symbol "let_fun " <|> "let_λ") >> letDecl) >> optSemic
17106 @[builtinTermParser] def «let_delayed» := leading_parser:leadPrec withPosition ("let_delayed" >> letDecl) >> optSemicolon termParser
17107

```

```

17108 def «scoped» := leading_parser "scoped "
17109 def «local»   := leading_parser "local "
17110 def attrKind  := leading_parser optional («scoped» <|> «local»)
17111 def attrInstance := ppGroup $ leading_parser attrKind >> attrParser
17112
17113 def attributes := leading_parser "@[" >> sepBy1 attrInstance ", " >> "]"
17114 def letRecDecl := leading_parser optional Command.docComment >> optional «attributes» >> letDecl
17115 def letRecDecls := leading_parser sepBy1 letRecDecl ", "
17116 @[builtinTermParser]
17117 def «letrec» := leading_parser:leadPrec withPosition (group ("let " >> nonReservedSymbol "rec ") >> letRecDecls) >> optSemicolon termPa
17118
17119 @[runBuiltinParserAttributeHooks]
17120 def whereDecls := leading_parser "where " >> many1Indent (group (letRecDecl >> optional ";"))
17121 @[runBuiltinParserAttributeHooks]
17122 def matchAltsWhereDecls := leading_parser matchAlts >> optional whereDecls
17123
17124 @[builtinTermParser] def noindex := leading_parser "no_index " >> termParser maxPrec
17125
17126 @[builtinTermParser] def binrel := leading_parser "binrel% " >> ident >> ppSpace >> termParser maxPrec >> termParser maxPrec
17127
17128 @[builtinTermParser] def forInMacro := leading_parser "forIn% " >> termParser maxPrec >> termParser maxPrec >> termParser maxPrec
17129
17130 @[builtinTermParser] def typeOf := leading_parser "typeOf% " >> termParser maxPrec
17131 @[builtinTermParser] def ensureTypeOf := leading_parser "ensureTypeOf% " >> termParser maxPrec >> strLit >> termParser
17132 @[builtinTermParser] def ensureExpectedType := leading_parser "ensureExpectedType% " >> strLit >> termParser maxPrec
17133 @[builtinTermParser] def noImplicitLambda := leading_parser "noImplicitLambda% " >> termParser maxPrec
17134
17135 def namedArgument := leading_parser atomic "(" >> ident >> " := " >> termParser >> ")"
17136 def ellipsis := leading_parser ".."
17137 def argument :=
17138   checkWsBefore "expected space" >>
17139   checkColGt "expected to be indented" >>
17140   (namedArgument <|> ellipsis <|> termParser argPrec)
17141 -- `app` precedence is `lead` (cannot be used as argument)
17142 -- `lhs` precedence is `max` (i.e. does not accept `arg` precedence)
17143 -- argument precedence is `arg` (i.e. does not accept `lead` precedence)
17144 @[builtinTermParser] def app := trailing_parser:leadPrec:maxPrec many1 argument
17145
17146 @[builtinTermParser] def proj := trailing_parser checkNoWsBefore >> "." >> checkNoWsBefore >> (fieldIdx <|> ident)
17147 @[builtinTermParser] def completion := trailing_parser checkNoWsBefore >> "."
17148 @[builtinTermParser] def arrayRef := trailing_parser checkNoWsBefore >> "[" >> termParser >> "]"
17149 @[builtinTermParser] def arrow := trailing_parser checkPrec 25 >> unicodeSymbol " → " " -> " >> termParser 25
17150
17151 def isIdent (stx : Syntax) : Bool :=
17152   -- antiquotations should also be allowed where an identifier is expected
17153   stx.isAntiquot || stx.isIdent
17154

```

```

17155 @[builtinTermParser] def explicitUniv : TrailingParser := trailing_parser checkStackTop isIdent "expected preceding identifier" >> chec
17156 @[builtinTermParser] def namedPattern : TrailingParser := trailing_parser checkStackTop isIdent "expected preceding identifier" >> chec
17157
17158 @[builtinTermParser] def pipeProj := trailing_parser:minPrec " |>." >> checkNoWsBefore >> (fieldIdx <|> ident) >> many argument
17159 @[builtinTermParser] def pipeCompletion := trailing_parser:minPrec " |>."
17160
17161 @[builtinTermParser] def subst := trailing_parser:75 " ▶ " >> sepBy1 (termParser 75) " ▶ "
17162
17163 -- NOTE: Doesn't call `categoryParser` directly in contrast to most other "static" quotations, so call `evalInsideQuot` explicitly
17164 @[builtinTermParser] def funBinder.quot : Parser := leading_parser "`(funBinder|" >> toggleInsideQuot (evalInsideQuot `funBinder funB
17165 def bracketedBinderF := bracketedBinder -- no default arg
17166 @[builtinTermParser] def bracketedBinder.quot : Parser := leading_parser "`(bracketedBinder|" >> toggleInsideQuot (evalInsideQuot `br
17167 @[builtinTermParser] def matchDiscr.quot : Parser := leading_parser "`(matchDiscr|" >> toggleInsideQuot (evalInsideQuot `matchDiscr m
17168 @[builtinTermParser] def attr.quot : Parser := leading_parser "`(attr|" >> toggleInsideQuot attrParser >> ")")
17169
17170 @[builtinTermParser] def panic := leading_parser:leadPrec "panic! " >> termParser
17171 @[builtinTermParser] def unreachable := leading_parser:leadPrec "unreachable!"
17172 @[builtinTermParser] def dbgTrace := leading_parser:leadPrec withPosition ("dbg_trace" >> ((interpolatedStr termParser) <|> termPars
17173 @[builtinTermParser] def assert := leading_parser:leadPrec withPosition ("assert! " >> termParser) >> optSemicolon termParser
17174
17175
17176 def macroArg := termParser maxPrec
17177 def macroDollarArg := leading_parser "$" >> termParser 10
17178 def macroLastArg := macroDollarArg <|> macroArg
17179
17180 -- Macro for avoiding exponentially big terms when using `STWorld`
17181 @[builtinTermParser] def stateRefT := leading_parser "StateRefT" >> macroArg >> macroLastArg
17182
17183 @[builtinTermParser] def dynamicQuot := leading_parser "(" >> ident >> "|" >> toggleInsideQuot (parserOfStack 1) >> ")"
17184
17185 end Term
17186
17187 @[builtinTermParser default+1] def Tactic.quot : Parser := leading_parser "`(tactic|" >> toggleInsideQuot tacticParser >> ")"
17188 @[builtinTermParser] def Tactic.quotSeq : Parser := leading_parser "`(tactic|" >> toggleInsideQuot Tactic.seq1 >> ")"
17189
17190 @[builtinTermParser] def Level.quot : Parser := leading_parser "`(level|" >> toggleInsideQuot levelParser >> ")"
17191
17192 builtin_initialize
17193   register_parser_alias "letDecl" Term.letDecl
17194   register_parser_alias "haveDecl" Term.haveDecl
17195   register_parser_alias "sufficesDecl" Term.sufficesDecl
17196   register_parser_alias "letRecDecls" Term.letRecDecls
17197   register_parser_alias "hole" Term.hole
17198   register_parser_alias "syntheticHole" Term.syntheticHole
17199   register_parser_alias "matchDiscr" Term.matchDiscr
17200   register_parser_alias "bracketedBinder" Term.bracketedBinder
17201   register_parser_alias "attrKind" Term.attrKind

```

17202

17203 end Parser

17204 end Lean