

```

1 // ::::::::::::::
2 // AbstractMVars.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.Meta.Basic
10
11 namespace Lean.Meta
12
13 structure AbstractMVarsResult where
14   paramNames : Array Name
15   numMVars    : Nat
16   expr        : Expr
17   deriving Inhabited, BEq
18
19 namespace AbstractMVars
20
21 open Std (HashMap)
22
23 structure State where
24   ngen      : NameGenerator
25   lctx      : LocalContext
26   nextParamIdx : Nat := 0
27   paramNames : Array Name := #[]
28   fvars      : Array Expr := #[]
29   lmap       : HashMap Name Level := {}
30   emap       : HashMap Name Expr  := {}
31
32 abbrev M := ReaderT MetavarContext (StateM State)
33
34 def mkFreshId : M Name := do
35   let s ← get
36   let fresh := s.ngen.curr
37   modify fun s => { s with ngen := s.ngen.next }
38   pure fresh
39
40 private partial def abstractLevelMVars (u : Level) : M Level := do
41   if !u.hasMVar then
42     return u
43   else
44     match u with
45     | Level.zero _      => return u
46     | Level.param _ _   => return u

```

```

47 | Level.succ v _      => return u.updateSucc! (← abstractLevelMVars v)
48 | Level.max v w _    => return u.updateMax! (← abstractLevelMVars v) (← abstractLevelMVars w)
49 | Level.imax v w _   => return u.updateIMax! (← abstractLevelMVars v) (← abstractLevelMVars w)
50 | Level.mvar mvarId _ =>
51   let mctx ← read
52   let depth := mctx.getLevelDepth mvarId;
53   if depth != mctx.depth then
54     return u -- metavariables from lower depths are treated as constants
55   else
56     let s ← get
57     match s.lmap.find? mvarId with
58     | some u => pure u
59     | none   =>
60       let paramId := Name.mkNum `_abstMVar s.nextParamIdx
61       let u := mkLevelParam paramId
62       modify fun s => { s with nextParamIdx := s.nextParamIdx + 1, lmap := s.lmap.insert mvarId u, paramNames := s.paramNames.push
paramId }
63       return u
64
65 partial def abstractExprMVars (e : Expr) : M Expr := do
66   if !e.hasMVar then
67     return e
68   else
69     match e with
70     | e@(Expr.lit _ _)      => return e
71     | e@(Expr.bvar _ _)    => return e
72     | e@(Expr.fvar _ _)    => return e
73     | e@(Expr.sort u _)    => return e.updateSort! (← abstractLevelMVars u)
74     | e@(Expr.const _ us _) => return e.updateConst! (← us.mapM abstractLevelMVars)
75     | e@(Expr.proj _ _ s _) => return e.updateProj! (← abstractExprMVars s)
76     | e@(Expr.app f a _)    => return e.updateApp! (← abstractExprMVars f) (← abstractExprMVars a)
77     | e@(Expr.mdata _ b _)  => return e.updateMData! (← abstractExprMVars b)
78     | e@(Expr.lam _ d b _)  => return e.updateLambdaE! (← abstractExprMVars d) (← abstractExprMVars b)
79     | e@(Expr.forallE _ d b _) => return e.updateForallE! (← abstractExprMVars d) (← abstractExprMVars b)
80     | e@(Expr.letE _ t v b _) => return e.updateLet! (← abstractExprMVars t) (← abstractExprMVars v) (← abstractExprMVars b)
81     | e@(Expr.mvar mvarId _) =>
82       let mctx ← read
83       let decl := mctx.getDecl mvarId
84       if decl.depth != mctx.depth then
85         return e
86       else
87         let s ← get
88         match s.emap.find? mvarId with
89         | some e =>
90           return e
91         | none   =>
92           let type ← abstractExprMVars decl.type

```

```

93     let fvarId ← mkFreshId
94     let fvar := mkFVar fvarId;
95     let userName := if decl.userName.isAnonymous then (`x).appendIndexAfter s.fvars.size else decl.userName
96     modify fun s => {
97       s with
98       emap := s.emap.insert mvarId fvar,
99       fvars := s.fvars.push fvar,
100       lctx := s.lctx.mkLocalDecl fvarId userName type }
101     return fvar
102
103 end AbstractMVars
104
105 /--
106   Abstract (current depth) metavariables occurring in `e`.
107   The result contains
108   - An array of universe level parameters that replaced universe metavariables occurring in `e`.
109   - The number of (expr) metavariables abstracted.
110   - And an expression of the form `fun (m_1 : A_1) ... (m_k : A_k) => e'`, where
111     `k` equal to the number of (expr) metavariables abstracted, and `e'` is `e` after we
112     replace the metavariables.
113
114   Example: given `f.{?u} ?m1` where `?m1 : ?m2 Nat`, `?m2 : Type -> Type`. This function returns
115   `{ levels := #[u], size := 2, expr := (fun (m2 : Type -> Type) (m1 : m2 Nat) => f.{u} m1) }`
116
117   This API can be used to "transport" to a different metavariable context.
118   Given a new metavariable context, we replace the `AbstractMVarsResult.levels` with
119   new fresh universe metavariables, and instantiate the `(m_i : A_i)` in the lambda-expression
120   with new fresh metavariables.
121
122   Application: we use this method to cache the results of type class resolution. -/
123 def abstractMVars (e : Expr) : MetaM AbstractMVarsResult := do
124   let e ← instantiateMVars e
125   let (e, s) := AbstractMVars.abstractExprMVars e (← getMCtx) { lctx := (← getLCtx), ngen := (← getNGen) }
126   setNGen s.ngen
127   let e := s.lctx.mkLambda s.fvars e
128   pure { paramNames := s.paramNames, numMVars := s.fvars.size, expr := e }
129
130 def openAbstractMVarsResult (a : AbstractMVarsResult) : MetaM (Array Expr × Array BinderInfo × Expr) := do
131   let us ← a.paramNames.mapM fun _ => mkFreshLevelMVar
132   let e := a.expr.instantiateLevelParamsArray a.paramNames us
133   lambdaMetaTelescope e (some a.numMVars)
134
135 end Lean.Meta
136 // ::::::::::::::
137 // AbstractNestedProofs.lean
138 // ::::::::::::::
139 /-

```

```

140 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
141 Released under Apache 2.0 license as described in the file LICENSE.
142 Authors: Leonardo de Moura
143 -/
144 import Lean.Meta.Closure
145
146 namespace Lean.Meta
147 namespace AbstractNestedProofs
148
149 def isNonTrivialProof (e : Expr) : MetaM Bool := do
150   if !(← isProof e) then
151     pure false
152   else
153     e.withApp fun f args =>
154       pure $ !f.isAtomic || args.any fun arg => !arg.isAtomic
155
156 structure Context where
157   baseName : Name
158
159 structure State where
160   nextIdx : Nat := 1
161
162 abbrev M := ReaderT Context $ MonadCacheT Expr Expr $ StateRefT State MetaM
163
164 private def mkAuxLemma (e : Expr) : M Expr := do
165   let ctx ← read
166   let s ← get
167   let lemmaName ← mkAuxName (ctx.baseName ++ `proof) s.nextIdx
168   modify fun s => { s with nextIdx := s.nextIdx + 1 }
169   mkAuxDefinitionFor lemmaName e
170
171 partial def visit (e : Expr) : M Expr := do
172   if e.isAtomic then
173     pure e
174   else
175     let visitBinders (xs : Array Expr) (k : M Expr) : M Expr := do
176       let localInstances ← getLocalInstances
177       let mut lctx ← getLCtx
178       for x in xs do
179         let xFVarId := x.fvarId!
180         let localDecl ← getLocalDecl xFVarId
181         let type ← visit localDecl.type
182         let localDecl := localDecl.setType type
183         let localDecl ← match localDecl.value? with
184           | some value => do let value ← visit value; pure $ localDecl.setValue value
185           | none      => pure localDecl
186       lctx := lctx.modifyLocalDecl xFVarId fun _ => localDecl

```

```

187     withLCtx lctx localInstances k
188     checkCache e fun _ => do
189       if (← isNonTrivialProof e) then
190         mkAuxLemma e
191       else match e with
192       | Expr.lam _ _ _ _ => lambdaLetTelescope e fun xs b => visitBinders xs do mkLambdaFVars xs (← visit b)
193       | Expr.letE _ _ _ _ => lambdaLetTelescope e fun xs b => visitBinders xs do mkLambdaFVars xs (← visit b)
194       | Expr.forallE _ _ _ _ => forallTelescope e fun xs b => visitBinders xs do mkForallFVars xs (← visit b)
195       | Expr.mdata _ b _ => return e.updateMData! (← visit b)
196       | Expr.proj _ _ b _ => return e.updateProj! (← visit b)
197       | Expr.app _ _ _ => e.withApp fun f args => return mkAppN f (← args.mapM visit)
198       | _ => pure e
199
200 end AbstractNestedProofs
201
202 /-- Replace proofs nested in `e` with new lemmas. The new lemmas have names of the form `mainDeclName.proof_<idx>` -/
203 def abstractNestedProofs (mainDeclName : Name) (e : Expr) : MetaM Expr :=
204   AbstractNestedProofs.visit e |>.run { baseName := mainDeclName } |>.run' { nextIdx := 1 }
205
206 end Lean.Meta
207 // ::::::::::::::
208 // AppBuilder.lean
209 // ::::::::::::::
210 /-
211 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
212 Released under Apache 2.0 license as described in the file LICENSE.
213 Authors: Leonardo de Moura
214 -/
215 import Lean.Structure
216 import Lean.Util.Recognizers
217 import Lean.Meta.SynthInstance
218 import Lean.Meta.Check
219
220 namespace Lean.Meta
221
222 /-- Return `id e` -/
223 def mkId (e : Expr) : MetaM Expr := do
224   let type ← inferType e
225   let u ← getLevel type
226   return mkApp2 (mkConst ``id [u]) type e
227
228 /-- Return `idRhs e` -/
229 def mkIdRhs (e : Expr) : MetaM Expr := do
230   let type ← inferType e
231   let u ← getLevel type
232   return mkApp2 (mkConst ``idRhs [u]) type e
233

```

```

234 /--
235   Given `e` s.t. `inferType e` is definitionally equal to `expectedType`, return
236   term `@id expectedType e`. -/
237 def mkExpectedTypeHint (e : Expr) (expectedType : Expr) : MetaM Expr := do
238   let u ← getLevel expectedType
239   return mkApp2 (mkConst ``id [u]) expectedType e
240
241 def mkEq (a b : Expr) : MetaM Expr := do
242   let aType ← inferType a
243   let u ← getLevel aType
244   return mkApp3 (mkConst ``Eq [u]) aType a b
245
246 def mkHEq (a b : Expr) : MetaM Expr := do
247   let aType ← inferType a
248   let bType ← inferType b
249   let u ← getLevel aType
250   return mkApp4 (mkConst ``HEq [u]) aType a bType b
251
252 def mkEqRefl (a : Expr) : MetaM Expr := do
253   let aType ← inferType a
254   let u ← getLevel aType
255   return mkApp2 (mkConst ``Eq.refl [u]) aType a
256
257 def mkHEqRefl (a : Expr) : MetaM Expr := do
258   let aType ← inferType a
259   let u ← getLevel aType
260   return mkApp2 (mkConst ``HEq.refl [u]) aType a
261
262 def mkAbsurd (e : Expr) (hp hnp : Expr) : MetaM Expr := do
263   let p ← inferType hp
264   let u ← getLevel e
265   return mkApp4 (mkConst ``absurd [u]) p e hp hnp
266
267 def mkFalseElim (e : Expr) (h : Expr) : MetaM Expr := do
268   let u ← getLevel e
269   return mkApp2 (mkConst ``False.elim [u]) e h
270
271 private def infer (h : Expr) : MetaM Expr := do
272   let hType ← inferType h
273   whnfD hType
274
275 private def hasTypeMsg (e type : Expr) : MessageData :=
276   m!"{indentExpr e}\nhas type{indentExpr type}"
277
278 private def throwAppBuilderException {α} (op : Name) (msg : MessageData) : MetaM α :=
279   throwError! "AppBuilder for '{op}', {msg}"
280

```

```

281 def mkEqSymm (h : Expr) : MetaM Expr := do
282   if h.isAppOf ``Eq.refl then
283     return h
284   else
285     let hType ← infer h
286     match hType.eq? with
287     | some ( $\alpha$ , a, b) =>
288       let u ← getLevel  $\alpha$ 
289       return mkApp4 (mkConst ``Eq.symm [u])  $\alpha$  a b h
290     | none => throwAppBuilderException ``Eq.symm ("equality proof expected" ++ hasTypeMsg h hType)
291
292 def mkEqTrans (h1 h2 : Expr) : MetaM Expr := do
293   if h1.isAppOf ``Eq.refl then
294     return h2
295   else if h2.isAppOf ``Eq.refl then
296     return h1
297   else
298     let hType1 ← infer h1
299     let hType2 ← infer h2
300     match hType1.eq?, hType2.eq? with
301     | some ( $\alpha$ , a, b), some (_, _, c) =>
302       let u ← getLevel  $\alpha$ 
303       return mkApp6 (mkConst ``Eq.trans [u])  $\alpha$  a b c h1 h2
304     | none, _ => throwAppBuilderException ``Eq.trans ("equality proof expected" ++ hasTypeMsg h1 hType1)
305     | _, none => throwAppBuilderException ``Eq.trans ("equality proof expected" ++ hasTypeMsg h2 hType2)
306
307 def mkHEqSymm (h : Expr) : MetaM Expr := do
308   if h.isAppOf ``HEq.refl then
309     return h
310   else
311     let hType ← infer h
312     match hType.heq? with
313     | some ( $\alpha$ , a,  $\beta$ , b) =>
314       let u ← getLevel  $\alpha$ 
315       return mkApp5 (mkConst ``HEq.symm [u])  $\alpha$   $\beta$  a b h
316     | none =>
317       throwAppBuilderException ``HEq.symm ("heterogeneous equality proof expected" ++ hasTypeMsg h hType)
318
319 def mkHEqTrans (h1 h2 : Expr) : MetaM Expr := do
320   if h1.isAppOf ``HEq.refl then
321     return h2
322   else if h2.isAppOf ``HEq.refl then
323     return h1
324   else
325     let hType1 ← infer h1
326     let hType2 ← infer h2
327     match hType1.heq?, hType2.heq? with

```

```

328 | some ( $\alpha$ , a,  $\beta$ , b), some (_, _,  $\gamma$ , c) =>
329   let u ← getLevel  $\alpha$ 
330   return mkApp8 (mkConst ``HEq.trans [u])  $\alpha$   $\beta$   $\gamma$  a b c h1 h2
331 | none, _ => throwAppBuilderException ``HEq.trans ("heterogeneous equality proof expected" ++ hasTypeMsg h1 hType1)
332 | _, none => throwAppBuilderException ``HEq.trans ("heterogeneous equality proof expected" ++ hasTypeMsg h2 hType2)
333
334 def mkEqOfHEq (h : Expr) : MetaM Expr := do
335   let hType ← infer h
336   match hType.heq? with
337   | some ( $\alpha$ , a,  $\beta$ , b) =>
338     unless (← isDefEq  $\alpha$   $\beta$ ) do
339       throwAppBuilderException ``eqOfHEq m!"heterogeneous equality types are not definitionally equal{indentExpr  $\alpha$ }\nis not
definitionally equal to{indentExpr  $\beta$ }"
340     let u ← getLevel  $\alpha$ 
341     return mkApp4 (mkConst ``eqOfHEq [u])  $\alpha$  a b h
342 | _ =>
343   throwAppBuilderException ``HEq.trans m!"heterogeneous equality proof expected{indentExpr h}"
344
345 def mkCongrArg (f h : Expr) : MetaM Expr := do
346   if h.isAppOf ``Eq.refl then
347     mkEqRefl (mkApp f h.appArg!)
348   else
349     let hType ← infer h
350     let fType ← infer f
351     match fType.arrow?, hType.eq? with
352     | some ( $\alpha$ ,  $\beta$ ), some (_, a, b) =>
353       let u ← getLevel  $\alpha$ 
354       let v ← getLevel  $\beta$ 
355       return mkApp6 (mkConst ``congrArg [u, v])  $\alpha$   $\beta$  a b f h
356     | none, _ => throwAppBuilderException ``congrArg ("non-dependent function expected" ++ hasTypeMsg f fType)
357     | _, none => throwAppBuilderException ``congrArg ("equality proof expected" ++ hasTypeMsg h hType)
358
359 def mkCongrFun (h a : Expr) : MetaM Expr := do
360   if h.isAppOf ``Eq.refl then
361     mkEqRefl (mkApp h.appArg! a)
362   else
363     let hType ← infer h
364     match hType.eq? with
365     | some ( $\rho$ , f, g) => do
366       let  $\rho$  ← whnfD  $\rho$ 
367       match  $\rho$  with
368       | Expr.forallE n  $\alpha$   $\beta$  _ =>
369         let  $\beta'$  := Lean.mkLambda n BinderInfo.default  $\alpha$   $\beta$ 
370         let u ← getLevel  $\alpha$ 
371         let v ← getLevel (mkApp  $\beta'$  a)
372         return mkApp6 (mkConst ``congrFun [u, v])  $\alpha$   $\beta'$  f g h a
373     | _ => throwAppBuilderException ``congrFun ("equality proof between functions expected" ++ hasTypeMsg h hType)

```



```

374 | _ => throwAppBuilderException ``congrFun ("equality proof expected" ++ hasTypeMsg h hType)
375
376 def mkCongr (h1 h2 : Expr) : MetaM Expr := do
377   if h1.isAppOf ``Eq.refl then
378     mkCongrArg h1.appArg! h2
379   else if h2.isAppOf ``Eq.refl then
380     mkCongrFun h1 h2.appArg!
381   else
382     let hType1 ← infer h1
383     let hType2 ← infer h2
384     match hType1.eq?, hType2.eq? with
385     | some (ρ, f, g), some (α, a, b) =>
386       let ρ ← whnfD ρ
387       match ρ.arrow? with
388       | some (_, β) => do
389         let u ← getLevel α
390         let v ← getLevel β
391         return mkApp8 (mkConst ``congr [u, v]) α β f g a b h1 h2
392       | _ => throwAppBuilderException ``congr ("non-dependent function expected" ++ hasTypeMsg h1 hType1)
393     | none, _ => throwAppBuilderException ``congr ("equality proof expected" ++ hasTypeMsg h1 hType1)
394     | _, none => throwAppBuilderException ``congr ("equality proof expected" ++ hasTypeMsg h2 hType2)
395
396 private def mkAppMFinal (methodName : Name) (f : Expr) (args : Array Expr) (instMVars : Array MVarId) : MetaM Expr := do
397   instMVars.forM fun mvarId => do
398     let mvarDecl ← getMVarDecl mvarId
399     let mvarVal ← synthInstance mvarDecl.type
400     assignExprMVar mvarId mvarVal
401   let result ← instantiateMVars (mkAppN f args)
402   if (← hasAssignableMVar result) then throwAppBuilderException methodName ("result contains metavariables" ++ indentExpr result)
403   return result
404
405 private partial def mkAppMArgs (f : Expr) (fType : Expr) (xs : Array Expr) : MetaM Expr :=
406   let rec loop (type : Expr) (i : Nat) (j : Nat) (args : Array Expr) (instMVars : Array MVarId) : MetaM Expr := do
407     if i >= xs.size then
408       mkAppMFinal `mkAppM f args instMVars
409     else match type with
410     | Expr.forallE n d b c =>
411       let d := d.instantiateRevRange j args.size args
412       match c.binderInfo with
413       | BinderInfo.implicit =>
414         let mvar ← mkFreshExprMVar d MetavarKind.natural n
415         loop b i j (args.push mvar) instMVars
416       | BinderInfo.instImplicit =>
417         let mvar ← mkFreshExprMVar d MetavarKind.synthetic n
418         loop b i j (args.push mvar) (instMVars.push mvar.mvarId!)
419     | _ =>
420       let x := xs[i]

```

```

421     let xType ← inferType x
422     if (← isDefEq d xType) then
423       loop b (i+1) j (args.push x) instMVars
424     else
425       throwAppTypeMismatch (mkAppN f args) x
426 | type =>
427   let type := type.instantiateRevRange j args.size args
428   let type ← whnfD type
429   if type.isForall then
430     loop type i args.size args instMVars
431   else
432     throwAppBuilderException `mkAppM m!"too many explicit arguments provided to{indentExpr f}\narguments{indentD xs}"
433 loop fType 0 0 #[] #[]
434
435 private def mkFun (constName : Name) : MetaM (Expr × Expr) := do
436   let cinfo ← getConstInfo constName
437   let us ← cinfo.levelParams.mapM fun _ => mkFreshLevelMVar
438   let f := mkConst constName us
439   let fType := cinfo.instantiateTypeLevelParams us
440   return (f, fType)
441
442 /--
443   Return the application `constName xs`.
444   It tries to fill the implicit arguments before the last element in `xs`.
445
446   Remark:
447   ``mkAppM `arbitrary #[α]`` returns `@arbitrary.{u} α` without synthesizing
448   the implicit argument occurring after `α`.
449   Given a `x : (([Decidable p] → Bool) × Nat`, ``mkAppM `Prod.fst #[x]`` returns `@Prod.fst ([Decidable p] → Bool) Nat x`
450 -/
451 def mkAppM (constName : Name) (xs : Array Expr) : MetaM Expr := do
452   traceCtx `Meta.appBuilder <| withNewMCtxDepth do
453     let (f, fType) ← mkFun constName
454     let r ← mkAppMArgs f fType xs
455     trace[Meta.appBuilder]! "constName: {constName}, xs: {xs}, result: {r}"
456     return r
457
458 private partial def mkAppOptMAux (f : Expr) (xs : Array (Option Expr)) : Nat → Array Expr → Nat → Array MVarId → Expr → MetaM Expr
459 | i, args, j, instMVars, Expr.forallE n d b c => do
460   let d := d.instantiateRevRange j args.size args
461   if h : i < xs.size then
462     match xs.get (i, h) with
463     | none =>
464       match c.binderInfo with
465       | BinderInfo.instImplicit => do
466         let mvar ← mkFreshExprMVar d MetavarKind.synthetic n
467         mkAppOptMAux f xs (i+1) (args.push mvar) j (instMVars.push mvar.mvarId!) b

```

```

468         | _ => do
469             let mvar ← mkFreshExprMVar d MetavarKind.natural n
470             mkAppOptMAux f xs (i+1) (args.push mvar) j instMVars b
471         | some x =>
472             let xType ← inferType x
473             if (← isDefEq d xType) then
474                 mkAppOptMAux f xs (i+1) (args.push x) j instMVars b
475             else
476                 throwAppTypeMismatch (mkAppN f args) x
477     else
478         mkAppMFinal `mkAppOptM f args instMVars
479 | i, args, j, instMVars, type => do
480     let type := type.instantiateRevRange j args.size args
481     let type ← whnfD type
482     if type.isForall then
483         mkAppOptMAux f xs i args args.size instMVars type
484     else if i == xs.size then
485         mkAppMFinal `mkAppOptM f args instMVars
486     else do
487         let xs : Array Expr := xs.foldl (fun r x? => match x? with | none => r | some x => r.push x) #[]
488         throwAppBuilderException `mkAppOptM ("too many arguments provided to" ++ indentExpr f ++ Format.line ++ "arguments" ++ xs)
489
490 /--
491 Similar to `mkAppM`, but it allows us to specify which arguments are provided explicitly using `Option` type.
492 Example:
493 Given `Pure.pure {m : Type u → Type v} [Pure m] {α : Type u} (a : α) : m α`,
494 ```
495 mkAppOptM `Pure.pure #[m, none, none, a]
496 ```
497 returns a `Pure.pure` application if the instance `Pure m` can be synthesized, and the universes match.
498 Note that,
499 ```
500 mkAppM `Pure.pure #[a]
501 ```
502 fails because the only explicit argument `(a : α)` is not sufficient for inferring the remaining arguments,
503 we would need the expected type. -/
504 def mkAppOptM (constName : Name) (xs : Array (Option Expr)) : MetaM Expr := do
505     traceCtx `Meta.appBuilder <| withNewMCtxDepth do
506         let (f, fType) ← mkFun constName
507         mkAppOptMAux f xs 0 #[] 0 #[] fType
508
509 def mkEqNDRec (motive h1 h2 : Expr) : MetaM Expr := do
510     if h2.isAppOf `Eq.refl then
511         return h1
512     else
513         let h2Type ← infer h2
514         match h2Type.eq? with

```

```

515 | none => throwAppBuilderException ``Eq.ndrec ("equality proof expected" ++ hasTypeMsg h2 h2Type)
516 | some ( $\alpha$ , a, b) =>
517   let u2 ← getLevel  $\alpha$ 
518   let motiveType ← infer motive
519   match motiveType with
520   | Expr.forallE _ _ (Expr.sort u1 _) _ =>
521     return mkAppN (mkConst ``Eq.ndrec [u1, u2]) #[ $\alpha$ , a, motive, h1, b, h2]
522   | _ => throwAppBuilderException ``Eq.ndrec ("invalid motive" ++ indentExpr motive)
523
524 def mkEqRec (motive h1 h2 : Expr) : MetaM Expr := do
525   if h2.isAppOf ``Eq.refl then
526     return h1
527   else
528     let h2Type ← infer h2
529     match h2Type.eq? with
530     | none => throwAppBuilderException ``Eq.rec ("equality proof expected" ++ indentExpr h2)
531     | some ( $\alpha$ , a, b) =>
532       let u2 ← getLevel  $\alpha$ 
533       let motiveType ← infer motive
534       match motiveType with
535       | Expr.forallE _ _ (Expr.forallE _ _ (Expr.sort u1 _) _) _ =>
536         return mkAppN (mkConst ``Eq.rec [u1, u2]) #[ $\alpha$ , a, motive, h1, b, h2]
537       | _ =>
538         throwAppBuilderException ``Eq.rec ("invalid motive" ++ indentExpr motive)
539
540 def mkEqMP (eqProof pr : Expr) : MetaM Expr :=
541   mkAppM ``Eq.mp #[eqProof, pr]
542
543 def mkEqMPR (eqProof pr : Expr) : MetaM Expr :=
544   mkAppM ``Eq.mpr #[eqProof, pr]
545
546 def mkNoConfusion (target : Expr) (h : Expr) : MetaM Expr := do
547   let type ← inferType h
548   let type ← whnf type
549   match type.eq? with
550   | none => throwAppBuilderException `noConfusion ("equality expected" ++ hasTypeMsg h type)
551   | some ( $\alpha$ , a, b) =>
552     let  $\alpha$  ← whnf  $\alpha$ 
553     matchConstInduct  $\alpha$ .getAppFn (fun _ => throwAppBuilderException `noConfusion ("inductive type expected" ++ indentExpr  $\alpha$ )) fun v us
554     => do
555       let u ← getLevel target
556       return mkAppN (mkConst (Name.mkStr v.name "noConfusion") (u :: us)) ( $\alpha$ .getAppArgs ++ #[target, a, b, h])
557
558 def mkPure (monad : Expr) (e : Expr) : MetaM Expr :=
559   mkAppOptM ``Pure.pure #[monad, none, none, e]
560 /--

```

```

561 `mkProjection s fieldName` return an expression for accessing field `fieldName` of the structure `s`.
562 Remark: `fieldName` may be a subfield of `s`. -/
563 partial def mkProjection : Expr → Name → MetaM Expr
564 | s, fieldName => do
565   let type ← inferType s
566   let type ← whnf type
567   match type.getAppFn with
568   | Expr.const structName us _ =>
569     let env ← getEnv
570     unless isStructureLike env structName do
571       throwAppBuilderException `mkProjection ("structure expected" ++ hasTypeMsg s type)
572     match getProjFnForField? env structName fieldName with
573     | some projFn =>
574       let params := type.getAppArgs
575       return mkApp (mkAppN (mkConst projFn us) params) s
576     | none =>
577       let fields := getStructureFields env structName
578       let r? ← fields.findSomeM? fun fieldName' => do
579         match isSubobjectField? env structName fieldName' with
580         | none => pure none
581         | some _ =>
582           let parent ← mkProjection s fieldName'
583           (do let r ← mkProjection parent fieldName'; return some r)
584         <|>
585       pure none
586     match r? with
587     | some r => pure r
588     | none => throwAppBuilderException `mkProjectionn ("invalid field name '" ++ toString fieldName ++ "' for" ++ hasTypeMsg s
type)
589   | _ => throwAppBuilderException `mkProjectionn ("structure expected" ++ hasTypeMsg s type)
590
591 private def mkListLitAux (nil : Expr) (cons : Expr) : List Expr → Expr
592 | [] => nil
593 | x::xs => mkApp (mkApp cons x) (mkListLitAux nil cons xs)
594
595 def mkListLit (type : Expr) (xs : List Expr) : MetaM Expr := do
596   let u ← getDecLevel type
597   let nil := mkApp (mkConst ``List.nil [u]) type
598   match xs with
599   | [] => return nil
600   | _ =>
601     let cons := mkApp (mkConst ``List.cons [u]) type
602     return mkListLitAux nil cons xs
603
604 def mkArrayLit (type : Expr) (xs : List Expr) : MetaM Expr := do
605   let u ← getDecLevel type
606   let listLit ← mkListLit type xs

```

```

607   return mkApp (mkApp (mkConst ``List.toArray [u]) type) listLit
608
609 def mkSorry (type : Expr) (synthetic : Bool) : MetaM Expr := do
610   let u ← getLevel type
611   return mkApp2 (mkConst ``sorryAx [u]) type (toExpr synthetic)
612
613 /-- Return `Decidable.decide p` -/
614 def mkDecide (p : Expr) : MetaM Expr :=
615   mkAppOptM ``Decidable.decide #[p, none]
616
617 /-- Return a proof for `p : Prop` using `decide p` -/
618 def mkDecideProof (p : Expr) : MetaM Expr := do
619   let decP      ← mkDecide p
620   let decEqTrue ← mkEq decP (mkConst ``Bool.true)
621   let h         ← mkEqRefl (mkConst ``Bool.true)
622   let h         ← mkExpectedTypeHint h decEqTrue
623   mkAppM ``ofDecideEqTrue #[h]
624
625 /-- Return `a < b` -/
626 def mkLt (a b : Expr) : MetaM Expr :=
627   mkAppM ``HasLess.Less #[a, b]
628
629 /-- Return `a <= b` -/
630 def mkLe (a b : Expr) : MetaM Expr :=
631   mkAppM ``HasLessEq.LessEq #[a, b]
632
633 /-- Return `arbitrary  $\alpha$ ` -/
634 def mkArbitrary ( $\alpha$  : Expr) : MetaM Expr :=
635   mkAppOptM ``arbitrary #[ $\alpha$ , none]
636
637 /-- Return `sorryAx type` -/
638 def mkSyntheticSorry (type : Expr) : MetaM Expr :=
639   return mkApp2 (mkConst ``sorryAx [← getLevel type]) type (mkConst ``Bool.true)
640
641 /-- Return `funext h` -/
642 def mkFunExt (h : Expr) : MetaM Expr :=
643   mkAppM ``funext #[h]
644
645 /-- Return `propext h` -/
646 def mkPropExt (h : Expr) : MetaM Expr :=
647   mkAppM ``propext #[h]
648
649 /-- Return `ofEqTrue h` -/
650 def mkOfEqTrue (h : Expr) : MetaM Expr :=
651   mkAppM ``ofEqTrue #[h]
652
653 /-- Return `eqTrue h` -/

```

```

654 def mkEqTrue (h : Expr) : MetaM Expr :=
655   mkAppM ``eqTrue #[h]
656
657 /--
658   Return `eqFalse h`
659   `h` must have type definitionally equal to ` $\neg p$ ` in the current
660   reducibility setting. -/
661 def mkEqFalse (h : Expr) : MetaM Expr :=
662   mkAppM ``eqFalse #[h]
663
664 /--
665   Return `eqFalse' h`
666   `h` must have type definitionally equal to ` $p \rightarrow \text{False}$ ` in the current
667   reducibility setting. -/
668 def mkEqFalse' (h : Expr) : MetaM Expr :=
669   mkAppM ``eqFalse' #[h]
670
671 def mkImpCongr (h1 h2 : Expr) : MetaM Expr :=
672   mkAppM ``impCongr #[h1, h2]
673
674 def mkImpCongrCtx (h1 h2 : Expr) : MetaM Expr :=
675   mkAppM ``impCongrCtx #[h1, h2]
676
677 def mkForallCongr (h : Expr) : MetaM Expr :=
678   mkAppM ``forallCongr #[h]
679
680 /-- Return instance for `[Monad m]` if there is one -/
681 def isMonad? (m : Expr) : MetaM (Option Expr) :=
682   try
683     let monadType ← mkAppM `Monad #[m]
684     let result ← trySynthInstance monadType
685     match result with
686     | LOption.some inst => pure inst
687     | _                  => pure none
688   catch _ =>
689     pure none
690
691 /-- Return `(n : type)`, a numeric literal of type `type`. The method fails if we don't have an instance `OfNat type n` -/
692 def mkNumeral (type : Expr) (n : Nat) : MetaM Expr := do
693   let u ← getDecLevel type
694   let inst ← synthInstance (mkApp2 (mkConst ``OfNat [u]) type (mkNatLit n))
695   return mkApp3 (mkConst ``OfNat.ofNat [u]) type (mkNatLit n) inst
696
697 /--
698   Return `a op b`, where `op` has name `opName` and is implemented using the typeclass `className`.
699   This method assumes `a` and `b` have the same type, and typeclass `className` is heterogeneous.
700   Examples of supported classes: `HAdd`, `HSub`, `HMul`.

```

```

701 We use heterogeneous operators to ensure we have a uniform representation.
702 -/
703 private def mkBinaryOp (className : Name) (opName : Name) (a b : Expr) : MetaM Expr := do
704   let aType ← inferType a
705   let u ← getDecLevel aType
706   let inst ← synthInstance (mkApp3 (mkConst className [u, u, u]) aType aType aType)
707   return mkApp6 (mkConst opName [u, u, u]) aType aType aType inst a b
708
709 /-- Return `a + b` using a heterogeneous `+`. This method assumes `a` and `b` have the same type. -/
710 def mkAdd (a b : Expr) : MetaM Expr := mkBinaryOp ``HAdd ``HAdd.hAdd a b
711
712 /-- Return `a - b` using a heterogeneous `-`. This method assumes `a` and `b` have the same type. -/
713 def mkSub (a b : Expr) : MetaM Expr := mkBinaryOp ``HSub ``HSub.hSub a b
714
715 /-- Return `a * b` using a heterogeneous `*`. This method assumes `a` and `b` have the same type. -/
716 def mkMul (a b : Expr) : MetaM Expr := mkBinaryOp ``HMul ``HMul.hMul a b
717
718 builtin_initialize registerTraceClass `Meta.appBuilder
719
720 end Lean.Meta
721 // ::::::::::::::
722 // Basic.lean
723 // ::::::::::::::
724 /-
725 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
726 Released under Apache 2.0 license as described in the file LICENSE.
727 Authors: Leonardo de Moura
728 -/
729 import Lean.Data.LOption
730 import Lean.Environment
731 import Lean.Class
732 import Lean.ReducibilityAttrs
733 import Lean.Util.Trace
734 import Lean.Util.RecDepth
735 import Lean.Util.PPExt
736 import Lean.Util.OccursCheck
737 import Lean.Compiler.InlineAttrs
738 import Lean.Meta.TransparencyMode
739 import Lean.Meta.DiscrTreeTypes
740 import Lean.Eval
741 import Lean.CoreM
742
743 /-
744 This module provides four (mutually dependent) goodies that are needed for building the elaborator and tactic frameworks.
745 1- Weak head normal form computation with support for metavariables and transparency modes.
746 2- Definitionally equality checking with support for metavariables (aka unification modulo definitional equality).
747 3- Type inference.

```



```

748 4- Type class resolution.
749
750 They are packed into the MetaM monad.
751 -/
752
753 namespace Lean.Meta
754
755 builtin_initialize isDefEqStuckExceptionId : InternalExceptionId ← registerInternalExceptionId `isDefEqStuck
756
757 structure Config where
758   foApprox      : Bool := false
759   ctxApprox     : Bool := false
760   quasiPatternApprox : Bool := false
761   /- When `constApprox` is set to true,
762     we solve `?m t == c` using
763     `?m := fun _ => c`
764     when `?m t` is not a higher-order pattern and `c` is not an application as -/
765   constApprox   : Bool := false
766   /-
767     When the following flag is set,
768     `isDefEq` throws the exception `Exception.isDefEqStuck`
769     whenever it encounters a constraint `?m ... == t` where
770     `?m` is read only.
771     This feature is useful for type class resolution where
772     we may want to notify the caller that the TC problem may be solveable
773     later after it assigns `?m`. -/
774   isDefEqStuckEx : Bool := false
775   transparency   : TransparencyMode := TransparencyMode.default
776   /- If zetaNonDep == false, then non dependent let-decls are not zeta expanded. -/
777   zetaNonDep     : Bool := true
778   /- When `trackZeta == true`, we store zetaFVarIds all free variables that have been zeta-expanded. -/
779   trackZeta     : Bool := false
780   unificationHints : Bool := true
781
782 structure ParamInfo where
783   implicit      : Bool := false
784   instImplicit  : Bool := false
785   hasFwdDeps   : Bool := false
786   backDeps     : Array Nat := #[]
787   deriving Inhabited
788
789 def ParamInfo.isExplicit (p : ParamInfo) : Bool :=
790   !p.implicit && !p.instImplicit
791
792 structure FunInfo where
793   paramInfo : Array ParamInfo := #[]
794   resultDeps : Array Nat := #[]

```

```

795
796 structure InfoCacheKey where
797   transparency : TransparencyMode
798   expr         : Expr
799   nargs?      : Option Nat
800   deriving Inhabited, BEq
801
802 namespace InfoCacheKey
803 instance : Hashable InfoCacheKey :=
804   (fun (transparency, expr, nargs) => mixHash (hash transparency) <| mixHash (hash expr) (hash nargs))
805 end InfoCacheKey
806
807 open Std (PersistentArray PersistentHashMap)
808
809 abbrev SynthInstanceCache := PersistentHashMap Expr (Option Expr)
810
811 abbrev InferTypeCache := PersistentExprStructMap Expr
812 abbrev FunInfoCache   := PersistentHashMap InfoCacheKey FunInfo
813 abbrev WhnfCache       := PersistentExprStructMap Expr
814 structure Cache where
815   inferType      : InferTypeCache := {}
816   funInfo        : FunInfoCache   := {}
817   synthInstance  : SynthInstanceCache := {}
818   whnfDefault    : WhnfCache := {} -- cache for closed terms and `TransparencyMode.default`
819   whnfAll        : WhnfCache := {} -- cache for closed terms and `TransparencyMode.all`
820   deriving Inhabited
821
822 structure PostponedEntry where
823   lhs : Level
824   rhs : Level
825
826 structure State where
827   mctx      : MetavarContext := {}
828   cache     : Cache := {}
829   /- When `trackZeta == true`, then any let-decl free variable that is zeta expansion performed by `MetaM` is stored in `zetaFVarIds`.
830   -/
831   zetaFVarIds : NameSet := {}
832   postponed   : PersistentArray PostponedEntry := {}
833   deriving Inhabited
834
835 structure Context where
836   config      : Config := {}
837   lctx        : LocalContext := {}
838   localInstances : LocalInstances := #[]
839
840 abbrev MetaM := ReaderT Context $ StateRefT State CoreM

```

```

841 instance : Inhabited (MetaM  $\alpha$ ) where
842   default := fun _ _ => arbitrary
843
844 instance : MonadLCtx MetaM where
845   getLCtx := return ( $\leftarrow$  read).lctx
846
847 instance : MonadMCtx MetaM where
848   getMCtx := return ( $\leftarrow$  get).mctx
849   modifyMCtx f := modify fun s => { s with mctx := f s.mctx }
850
851 instance : AddMessageContext MetaM where
852   addMessageContext := addMessageContextFull
853
854 @[inline] def MetaM.run (x : MetaM  $\alpha$ ) (ctx : Context := {}) (s : State := {}) : CoreM ( $\alpha \times$  State) :=
855   x ctx |>.run s
856
857 @[inline] def MetaM.run' (x : MetaM  $\alpha$ ) (ctx : Context := {}) (s : State := {}) : CoreM  $\alpha$  :=
858   Prod.fst <$> x.run ctx s
859
860 @[inline] def MetaM.toIO (x : MetaM  $\alpha$ ) (ctxCore : Core.Context) (sCore : Core.State) (ctx : Context := {}) (s : State := {}) : IO ( $\alpha \times$ 
Core.State  $\times$  State) := do
861   let ((a, s), sCore)  $\leftarrow$  (x.run ctx s).toIO ctxCore sCore
862   pure (a, sCore, s)
863
864 instance [MetaEval  $\alpha$ ] : MetaEval (MetaM  $\alpha$ ) :=
865   {fun env opts x _ => MetaEval.eval env opts x.run' true}
866
867 protected def throwIsDefEqStuck { $\alpha$ } : MetaM  $\alpha$  :=
868   throw <| Exception.internal isDefEqStuckExceptionId
869
870 builtin_initialize
871   registerTraceClass `Meta
872   registerTraceClass `Meta.debug
873
874 @[inline] def liftMetaM [MonadLiftT MetaM m] (x : MetaM  $\alpha$ ) : m  $\alpha$  :=
875   liftM x
876
877 @[inline] def mapMetaM [MonadControlT MetaM m] [Monad m] (f : forall { $\alpha$ }, MetaM  $\alpha$   $\rightarrow$  MetaM  $\alpha$ ) { $\alpha$ } (x : m  $\alpha$ ) : m  $\alpha$  :=
878   controlAt MetaM fun runInBase => f <| runInBase x
879
880 @[inline] def map1MetaM [MonadControlT MetaM m] [Monad m] (f : forall { $\alpha$ }, ( $\beta \rightarrow$  MetaM  $\alpha$ )  $\rightarrow$  MetaM  $\alpha$ ) { $\alpha$ } (k :  $\beta \rightarrow$  m  $\alpha$ ) : m  $\alpha$  :=
881   controlAt MetaM fun runInBase => f fun b => runInBase <| k b
882
883 @[inline] def map2MetaM [MonadControlT MetaM m] [Monad m] (f : forall { $\alpha$ }, ( $\beta \rightarrow \gamma \rightarrow$  MetaM  $\alpha$ )  $\rightarrow$  MetaM  $\alpha$ ) { $\alpha$ } (k :  $\beta \rightarrow \gamma \rightarrow$  m  $\alpha$ ) : m  $\alpha$  :=
884   controlAt MetaM fun runInBase => f fun b c => runInBase <| k b c
885
886 section Methods

```

```

887 variable [MonadControlT MetaM n] [Monad n]
888
889 @[inline] def modifyCache (f : Cache → Cache) : MetaM Unit :=
890   modify fun ⟨mctx, cache, zetaFVarIds, postponed⟩ => ⟨mctx, f cache, zetaFVarIds, postponed⟩
891
892 @[inline] def modifyInferTypeCache (f : InferTypeCache → InferTypeCache) : MetaM Unit :=
893   modifyCache fun ⟨ic, c1, c2, c3, c4⟩ => ⟨f ic, c1, c2, c3, c4⟩
894
895 def getLocalInstances : MetaM LocalInstances :=
896   return (← read).localInstances
897
898 def getConfig : MetaM Config :=
899   return (← read).config
900
901 def setMCtx (mctx : MetavarContext) : MetaM Unit :=
902   modify fun s => { s with mctx := mctx }
903
904 def resetZetaFVarIds : MetaM Unit :=
905   modify fun s => { s with zetaFVarIds := {} }
906
907 def getZetaFVarIds : MetaM NameSet :=
908   return (← get).zetaFVarIds
909
910 def getPostponed : MetaM (PersistentArray PostponedEntry) :=
911   return (← get).postponed
912
913 def setPostponed (postponed : PersistentArray PostponedEntry) : MetaM Unit :=
914   modify fun s => { s with postponed := postponed }
915
916 @[inline] def modifyPostponed (f : PersistentArray PostponedEntry → PersistentArray PostponedEntry) : MetaM Unit :=
917   modify fun s => { s with postponed := f s.postponed }
918
919 builtin_initialize whnfRef : IO.Ref (Expr → MetaM Expr) ← IO.mkRef fun _ => throwError "whnf implementation was not set"
920 builtin_initialize inferTypeRef : IO.Ref (Expr → MetaM Expr) ← IO.mkRef fun _ => throwError "inferType implementation was not set"
921 builtin_initialize isExprDefEqAuxRef : IO.Ref (Expr → Expr → MetaM Bool) ← IO.mkRef fun _ _ => throwError "isDefEq implementation was
not set"
922 builtin_initialize synthPendingRef : IO.Ref (MVarId → MetaM Bool) ← IO.mkRef fun _ => pure false
923
924 def whnf (e : Expr) : MetaM Expr :=
925   withIncRecDepth do (← whnfRef.get) e
926
927 def whnfForall (e : Expr) : MetaM Expr := do
928   let e' ← whnf e
929   if e'.isForall then pure e' else pure e
930
931 def inferType (e : Expr) : MetaM Expr :=
932   withIncRecDepth do (← inferTypeRef.get) e

```

```

933
934 protected def isExprDefEqAux (t s : Expr) : MetaM Bool :=
935   withIncRecDepth do (← isExprDefEqAuxRef.get) t s
936
937 protected def synthPending (mvarId : MVarId) : MetaM Bool :=
938   withIncRecDepth do (← synthPendingRef.get) mvarId
939
940 -- withIncRecDepth for a monad `n` such that `[MonadControlT MetaM n]`
941 protected def withIncRecDepth {α} (x : n α) : n α :=
942   mapMetaM (withIncRecDepth (m := MetaM)) x
943
944 private def mkFreshExprMVarAtCore
945   (mvarId : MVarId) (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr) (kind : MetavarKind) (userName : Name)
946   (numScopeArgs : Nat) : MetaM Expr := do
947     modifyMCtx fun mctx => mctx.addExprMVarDecl mvarId userName lctx localInsts type kind numScopeArgs;
948     return mkMVar mvarId
949
950 def mkFreshExprMVarAt
951   (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr)
952   (kind : MetavarKind := MetavarKind.natural) (userName : Name := Name.anonymous) (numScopeArgs : Nat := 0)
953   : MetaM Expr := do
954   let mvarId ← mkFreshId
955   mkFreshExprMVarAtCore mvarId lctx localInsts type kind userName numScopeArgs
956
957 def mkFreshLevelMVar : MetaM Level := do
958   let mvarId ← mkFreshId
959   modifyMCtx fun mctx => mctx.addLevelMVarDecl mvarId;
960   return mkLevelMVar mvarId
961
962 private def mkFreshExprMVarCore (type : Expr) (kind : MetavarKind) (userName : Name) : MetaM Expr := do
963   let lctx ← getLCtx
964   let localInsts ← getLocalInstances
965   mkFreshExprMVarAt lctx localInsts type kind userName
966
967 private def mkFreshExprMVarImpl (type? : Option Expr) (kind : MetavarKind) (userName : Name) : MetaM Expr :=
968   match type? with
969   | some type => mkFreshExprMVarCore type kind userName
970   | none      => do
971     let u ← mkFreshLevelMVar
972     let type ← mkFreshExprMVarCore (mkSort u) MetavarKind.natural Name.anonymous
973     mkFreshExprMVarCore type kind userName
974
975 def mkFreshExprMVar (type? : Option Expr) (kind := MetavarKind.natural) (userName := Name.anonymous) : MetaM Expr :=
976   mkFreshExprMVarImpl type? kind userName
977
978 def mkFreshTypeMVar (kind := MetavarKind.natural) (userName := Name.anonymous) : MetaM Expr := do
979   let u ← mkFreshLevelMVar

```

```

979   mkFreshExprMVar (mkSort u) kind userName
980
981 /- Low-level version of `MkFreshExprMVar` which allows users to create/reserve a `mvarId` using `mkFreshId`, and then later create
982   the metavar using this method. -/
983 private def mkFreshExprMVarWithIdCore (mvarId : MVarId) (type : Expr)
984   (kind : MetavarKind := MetavarKind.natural) (userName : Name := Name.anonymous) (numScopeArgs : Nat := 0)
985   : MetaM Expr := do
986   let lctx ← getLCtx
987   let localInsts ← getLocalInstances
988   mkFreshExprMVarAtCore mvarId lctx localInsts type kind userName numScopeArgs
989
990 def mkFreshExprMVarWithId (mvarId : MVarId) (type? : Option Expr := none) (kind : MetavarKind := MetavarKind.natural) (userName :=
Name.anonymous) : MetaM Expr :=
991   match type? with
992   | some type => mkFreshExprMVarWithIdCore mvarId type kind userName
993   | none      => do
994     let u ← mkFreshLevelMVar
995     let type ← mkFreshExprMVar (mkSort u)
996     mkFreshExprMVarWithIdCore mvarId type kind userName
997
998 def getTransparency : MetaM TransparencyMode :=
999   return (← getConfig).transparency
1000
1001 def shouldReduceAll : MetaM Bool :=
1002   return (← getTransparency) == TransparencyMode.all
1003
1004 def shouldReduceReducibleOnly : MetaM Bool :=
1005   return (← getTransparency) == TransparencyMode.reducible
1006
1007 def getMVarDecl (mvarId : MVarId) : MetaM MetavarDecl := do
1008   let mctx ← getMCtx
1009   match mctx.findDecl? mvarId with
1010   | some d => pure d
1011   | none   => throwError! "unknown metavariable '{mkMVar mvarId}'"
1012
1013 def setMVarKind (mvarId : MVarId) (kind : MetavarKind) : MetaM Unit :=
1014   modifyMCtx fun mctx => mctx.setMVarKind mvarId kind
1015
1016 /- Update the type of the given metavariable. This function assumes the new type is
1017   definitionally equal to the current one -/
1018 def setMVarType (mvarId : MVarId) (type : Expr) : MetaM Unit := do
1019   modifyMCtx fun mctx => mctx.setMVarType mvarId type
1020
1021 def isReadOnlyExprMVar (mvarId : MVarId) : MetaM Bool := do
1022   let mvarDecl ← getMVarDecl mvarId
1023   let mctx      ← getMCtx
1024   return mvarDecl.depth != mctx.depth

```

```

1025
1026 def isReadOnlyOrSyntheticOpaqueExprMVar (mvarId : MVarId) : MetaM Bool := do
1027   let mvarDecl ← getMVarDecl mvarId
1028   match mvarDecl.kind with
1029   | MetavarKind.syntheticOpaque => pure true
1030   | _ =>
1031     let mctx ← getMCtx
1032     return mvarDecl.depth != mctx.depth
1033
1034 def isReadOnlyLevelMVar (mvarId : MVarId) : MetaM Bool := do
1035   let mctx ← getMCtx
1036   match mctx.findLevelDepth? mvarId with
1037   | some depth => return depth != mctx.depth
1038   | _          => throwError! "unknown universe metavariable '{mkLevelMVar mvarId}'"
1039
1040 def renameMVar (mvarId : MVarId) (newUserName : Name) : MetaM Unit :=
1041   modifyMCtx fun mctx => mctx.renameMVar mvarId newUserName
1042
1043 def isExprMVarAssigned (mvarId : MVarId) : MetaM Bool :=
1044   return (← getMCtx).isExprAssigned mvarId
1045
1046 def getExprMVarAssignment? (mvarId : MVarId) : MetaM (Option Expr) :=
1047   return (← getMCtx).getExprAssignment? mvarId
1048
1049 /-- Return true if `e` contains `mvarId` directly or indirectly -/
1050 def occursCheck (mvarId : MVarId) (e : Expr) : MetaM Bool :=
1051   return (← getMCtx).occursCheck mvarId e
1052
1053 def assignExprMVar (mvarId : MVarId) (val : Expr) : MetaM Unit :=
1054   modifyMCtx fun mctx => mctx.assignExpr mvarId val
1055
1056 def isDelayedAssigned (mvarId : MVarId) : MetaM Bool :=
1057   return (← getMCtx).isDelayedAssigned mvarId
1058
1059 def getDelayedAssignment? (mvarId : MVarId) : MetaM (Option DelayedMetavarAssignment) :=
1060   return (← getMCtx).getDelayedAssignment? mvarId
1061
1062 def hasAssignableMVar (e : Expr) : MetaM Bool :=
1063   return (← getMCtx).hasAssignableMVar e
1064
1065 def throwUnknownFVar {α} (fvarId : FVarId) : MetaM α :=
1066   throwError! "unknown free variable '{mkFVar fvarId}'"
1067
1068 def findLocalDecl? (fvarId : FVarId) : MetaM (Option LocalDecl) :=
1069   return (← getLCtx).find? fvarId
1070
1071 def getLocalDecl (fvarId : FVarId) : MetaM LocalDecl := do

```

```

1072 match (← getLCtx).find? fvarId with
1073 | some d => pure d
1074 | none   => throwUnknownFVar fvarId
1075
1076 def getFVarLocalDecl (fvar : Expr) : MetaM LocalDecl :=
1077   getLocalDecl fvar.fvarId!
1078
1079 def getLocalDeclFromUserName (userName : Name) : MetaM LocalDecl := do
1080   match (← getLCtx).findFromUserName? userName with
1081   | some d => pure d
1082   | none   => throwError! "unknown local declaration '{userName}'"
1083
1084 def instantiateLevelMVars (u : Level) : MetaM Level :=
1085   MetavarContext.instantiateLevelMVars u
1086
1087 def instantiateMVars (e : Expr) : MetaM Expr :=
1088   (MetavarContext.instantiateExprMVars e).run
1089
1090 def instantiateLocalDeclMVars (localDecl : LocalDecl) : MetaM LocalDecl := do
1091   match localDecl with
1092   | LocalDecl.cdecl idx id n type bi =>
1093     let type ← instantiateMVars type
1094     return LocalDecl.cdecl idx id n type bi
1095   | LocalDecl.ldecl idx id n type val nonDep =>
1096     let type ← instantiateMVars type
1097     let val   ← instantiateMVars val
1098     return LocalDecl.ldecl idx id n type val nonDep
1099
1100 @[inline] def liftMkBindingM {α} (x : MetavarContext.MkBindingM α) : MetaM α := do
1101   match x (← getLCtx) { mctx := (← getMCtx), ngen := (← getNGen) } with
1102   | EStateM.Result.ok e newS => do
1103     setNGen newS.ngen;
1104     setMCtx newS.mctx;
1105     pure e
1106   | EStateM.Result.error (MetavarContext.MkBinding.Exception.revertFailure mctx lctx toRevert decl) newS => do
1107     setMCtx newS.mctx;
1108     setNGen newS.ngen;
1109     throwError "failed to create binder due to failure when reverting variable dependencies"
1110
1111 def mkForallFVars (xs : Array Expr) (e : Expr) (usedOnly : Bool := false) (usedLetOnly : Bool := true) : MetaM Expr :=
1112   if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.mkForall xs e usedOnly usedLetOnly
1113
1114 def mkLambdaFVars (xs : Array Expr) (e : Expr) (usedOnly : Bool := false) (usedLetOnly : Bool := true) : MetaM Expr :=
1115   if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.mkLambda xs e usedOnly usedLetOnly
1116
1117 def mkLetFVars (xs : Array Expr) (e : Expr) : MetaM Expr :=
1118   mkLambdaFVars xs e

```



```

1119
1120 def mkArrow (d b : Expr) : MetaM Expr := do
1121   let n ← mkFreshUserName `x
1122   return Lean.mkForall n BinderInfo.default d b
1123
1124 def elimMVarDeps (xs : Array Expr) (e : Expr) (preserveOrder : Bool := false) : MetaM Expr :=
1125   if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.elimMVarDeps xs e preserveOrder
1126
1127 @[inline] def withConfig {α} (f : Config → Config) : n α → n α :=
1128   mapMetaM <| withReader (fun ctx => { ctx with config := f ctx.config })
1129
1130 @[inline] def withTrackingZeta {α} (x : n α) : n α :=
1131   withConfig (fun cfg => { cfg with trackZeta := true }) x
1132
1133 @[inline] def withTransparency {α} (mode : TransparencyMode) : n α → n α :=
1134   mapMetaM <| withConfig (fun config => { config with transparency := mode })
1135
1136 @[inline] def withDefault {α} (x : n α) : n α :=
1137   withTransparency TransparencyMode.default x
1138
1139 @[inline] def withReducible {α} (x : n α) : n α :=
1140   withTransparency TransparencyMode.reducible x
1141
1142 @[inline] def withReducibleAndInstances {α} (x : n α) : n α :=
1143   withTransparency TransparencyMode.instances x
1144
1145 @[inline] def withAtLeastTransparency {α} (mode : TransparencyMode) (x : n α) : n α :=
1146   withConfig
1147     (fun config =>
1148       let oldMode := config.transparency
1149       let mode     := if oldMode.lt mode then mode else oldMode
1150       { config with transparency := mode })
1151   x
1152
1153 /-- Save cache, execute `x`, restore cache -/
1154 @[inline] private def savingCacheImpl {α} (x : MetaM α) : MetaM α := do
1155   let s ← get
1156   let savedCache := s.cache
1157   try x finally modify fun s => { s with cache := savedCache }
1158
1159 @[inline] def savingCache {α} : n α → n α :=
1160   mapMetaM savingCacheImpl
1161
1162 def getTheoremInfo (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
1163   if (← shouldReduceAll) then
1164     return some info
1165   else

```

```

1166     return none
1167
1168 private def getDefInfoTemp (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
1169   match (← getTransparency) with
1170   | TransparencyMode.all => return some info
1171   | TransparencyMode.default => return some info
1172   | _ =>
1173     if (← isReducible info.name) then
1174       return some info
1175     else
1176       return none
1177
1178 /- Remark: we later define `getConst?` at `GetConst.lean` after we define `Instances.lean`.
1179    This method is only used to implement `isClassQuickConst?`.
1180    It is very similar to `getConst?`, but it returns none when `TransparencyMode.instances` and
1181    `constName` is an instance. This difference should be irrelevant for `isClassQuickConst?`. -/
1182 private def getConstTemp? (constName : Name) : MetaM (Option ConstantInfo) := do
1183   let env ← getEnv
1184   match env.find? constName with
1185   | some (info@(ConstantInfo.thmInfo _)) => getTheoremInfo info
1186   | some (info@(ConstantInfo.defnInfo _)) => getDefInfoTemp info
1187   | some info => pure (some info)
1188   | none => throwUnknownConstant constName
1189
1190 private def isClassQuickConst? (constName : Name) : MetaM (LOption Name) := do
1191   let env ← getEnv
1192   if isClass env constName then
1193     pure (LOption.some constName)
1194   else
1195     match (← getConstTemp? constName) with
1196     | some _ => pure LOption.undef
1197     | none => pure LOption.none
1198
1199 private partial def isClassQuick? : Expr → MetaM (LOption Name)
1200 | Expr.bvar ..      => pure LOption.none
1201 | Expr.lit ..       => pure LOption.none
1202 | Expr.fvar ..      => pure LOption.none
1203 | Expr.sort ..      => pure LOption.none
1204 | Expr.lam ..       => pure LOption.none
1205 | Expr.letE ..      => pure LOption.undef
1206 | Expr.proj ..      => pure LOption.undef
1207 | Expr.forallE _ _ b _ => isClassQuick? b
1208 | Expr.mdata _ e _   => isClassQuick? e
1209 | Expr.const n _ _   => isClassQuickConst? n
1210 | Expr.mvar mvarId _ => do
1211   match (← getExprMVarAssignment? mvarId) with
1212   | some val => isClassQuick? val

```

```

1213 | none      => pure LOption.none
1214 | Expr.app f _ _      =>
1215   match f.getAppFn with
1216   | Expr.const n .. => isClassQuickConst? n
1217   | Expr.lam ..     => pure LOption.undef
1218   | _               => pure LOption.none
1219
1220 def saveAndResetSynthInstanceCache : MetaM SynthInstanceCache := do
1221   let s ← get
1222   let savedSythInstance := s.cache.synthInstance
1223   modifyCache fun c => { c with synthInstance := {} }
1224   pure savedSythInstance
1225
1226 def restoreSynthInstanceCache (cache : SynthInstanceCache) : MetaM Unit :=
1227   modifyCache fun c => { c with synthInstance := cache }
1228
1229 @[inline] private def resettingSynthInstanceCacheImpl {α} (x : MetaM α) : MetaM α := do
1230   let savedSythInstance ← saveAndResetSynthInstanceCache
1231   try x finally restoreSynthInstanceCache savedSythInstance
1232
1233 /-- Reset `synthInstance` cache, execute `x`, and restore cache -/
1234 @[inline] def resettingSynthInstanceCache {α} : n α → n α :=
1235   mapMetaM resettingSynthInstanceCacheImpl
1236
1237 @[inline] def resettingSynthInstanceCacheWhen {α} (b : Bool) (x : n α) : n α :=
1238   if b then resettingSynthInstanceCache x else x
1239
1240 private def withNewLocalInstanceImp {α} (className : Name) (fvar : Expr) (k : MetaM α) : MetaM α := do
1241   let localDecl ← getFVarLocalDecl fvar
1242   /- Recall that we use `auxDecl` binderInfo when compiling recursive declarations. -/
1243   match localDecl.binderInfo with
1244   | BinderInfo.auxDecl => k
1245   | _ =>
1246     resettingSynthInstanceCache <|
1247       withReader
1248         (fun ctx => { ctx with localInstances := ctx.localInstances.push { className := className, fvar := fvar } })
1249         k
1250
1251 /-- Add entry `{ className := className, fvar := fvar }` to localInstances,
1252   and then execute continuation `k`.
1253   It resets the type class cache using `resettingSynthInstanceCache`. -/
1254 def withNewLocalInstance {α} (className : Name) (fvar : Expr) : n α → n α :=
1255   mapMetaM <| withNewLocalInstanceImp className fvar
1256
1257 private def fvarsSizeLtMaxFVars (fvars : Array Expr) (maxFVars? : Option Nat) : Bool :=
1258   match maxFVars? with
1259   | some maxFVars => fvars.size < maxFVars

```

```

1260 | none          => true
1261
1262 mutual
1263 /--
1264   `withNewLocalInstances isClassExpensive fvars j k` updates the vector or local instances
1265   using free variables `fvars[j] ... fvars.back`, and execute `k`.
1266
1267   - `isClassExpensive` is defined later.
1268   - The type class cache is reset whenever a new local instance is found.
1269   - `isClassExpensive` uses `whnf` which depends (indirectly) on the set of local instances.
1270   Thus, each new local instance requires a new `resettingSynthInstanceCache`. -/
1271 private partial def withNewLocalInstancesImp {α}
1272   (fvars : Array Expr) (i : Nat) (k : MetaM α) : MetaM α := do
1273   if h : i < fvars.size then
1274     let fvar := fvars.get (i, h)
1275     let decl ← getFVarLocalDecl fvar
1276     match (← isClassQuick? decl.type) with
1277     | LOption.none    => withNewLocalInstancesImp fvars (i+1) k
1278     | LOption.undef  =>
1279       match (← isClassExpensive? decl.type) with
1280       | none    => withNewLocalInstancesImp fvars (i+1) k
1281       | some c => withNewLocalInstance c fvar <| withNewLocalInstancesImp fvars (i+1) k
1282     | LOption.some c => withNewLocalInstance c fvar <| withNewLocalInstancesImp fvars (i+1) k
1283   else
1284     k
1285
1286 /--
1287   `forallTelescopeAuxAux lctx fvars j type`
1288   Remarks:
1289   - `lctx` is the `MetaM` local context extended with declarations for `fvars`.
1290   - `type` is the type we are computing the telescope for. It contains only
1291     dangling bound variables in the range `[j, fvars.size)`
1292   - if `reducing? == true` and `type` is not `forallE`, we use `whnf`.
1293   - when `type` is not a `forallE` nor it can't be reduced to one, we
1294     excute the continuation `k`.
1295
1296   Here is an example that demonstrates the `reducing?`.
1297   Suppose we have
1298   ```
1299   abbrev StateM s a := s -> Prod a s
1300   ```
1301   Now, assume we are trying to build the telescope for
1302   ```
1303   forall (x : Nat), StateM Int Bool
1304   ```
1305   if `reducing == true`, the function executes `k #[(x : Nat) (s : Int)] Bool`.
1306   if `reducing == false`, the function executes `k #[(x : Nat)] (StateM Int Bool)`

```

```

1307
1308   if `maxFVars?` is `some max`, then we interrupt the telescope construction
1309   when `fvars.size == max`
1310   -/
1311   private partial def forallTelescopeReducingAuxAux {α}
1312     (reducing      : Bool) (maxFVars? : Option Nat)
1313     (type          : Expr)
1314     (k             : Array Expr → Expr → MetaM α) : MetaM α := do
1315     let rec process (lctx : LocalContext) (fvars : Array Expr) (j : Nat) (type : Expr) : MetaM α := do
1316       match type with
1317       | Expr.forallE n d b c =>
1318         if fvarsSizeLtMaxFVars fvars maxFVars? then
1319           let d      := d.instantiateRevRange j fvars.size fvars
1320           let fvarId ← mkFreshId
1321           let lctx   := lctx.mkLocalDecl fvarId n d c.binderInfo
1322           let fvar   := mkFVar fvarId
1323           let fvars  := fvars.push fvar
1324           process lctx fvars j b
1325         else
1326           let type := type.instantiateRevRange j fvars.size fvars;
1327           withReader (fun ctx => { ctx with lctx := lctx }) do
1328             withNewLocalInstancesImp fvars j do
1329               k fvars type
1330       | _ =>
1331         let type := type.instantiateRevRange j fvars.size fvars;
1332         withReader (fun ctx => { ctx with lctx := lctx }) do
1333           withNewLocalInstancesImp fvars j do
1334             if reducing && fvarsSizeLtMaxFVars fvars maxFVars? then
1335               let newType ← whnf type
1336               if newType.isForall then
1337                 process lctx fvars fvars.size newType
1338               else
1339                 k fvars type
1340             else
1341               k fvars type
1342         process (← getLCtx) #[] 0 type
1343
1344   private partial def forallTelescopeReducingAux {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → MetaM α) : MetaM α
1345   := do
1346     match maxFVars? with
1347     | some 0 => k #[] type
1348     | _ => do
1349       let newType ← whnf type
1350       if newType.isForall then
1351         forallTelescopeReducingAuxAux true maxFVars? newType k
1352       else
1353         k #[] type

```

```

1353
1354 private partial def isClassExpensive? : Expr → MetaM (Option Name)
1355   | type => withReducible <| -- when testing whether a type is a type class, we only unfold reducible constants.
1356     forallTelescopeReducingAux type none fun xs type => do
1357       let env ← getEnv
1358       match type.getAppFn with
1359       | Expr.const c _ _ => do
1360         if isClass env c then
1361           return some c
1362         else
1363           -- make sure abbreviations are unfolded
1364           match (← whnf type).getAppFn with
1365           | Expr.const c _ _ => return if isClass env c then some c else none
1366           | _ => return none
1367       | _ => return none
1368
1369 private partial def isClassImp? (type : Expr) : MetaM (Option Name) := do
1370   match (← isClassQuick? type) with
1371   | LOption.none    => pure none
1372   | LOption.some c => pure (some c)
1373   | LOption.undef  => isClassExpensive? type
1374
1375 end
1376
1377 def isClass? (type : Expr) : MetaM (Option Name) :=
1378   try isClassImp? type catch _ => pure none
1379
1380 private def withNewLocalInstancesImpAux {α} (fvars : Array Expr) (j : Nat) : n α → n α :=
1381   mapMetaM <| withNewLocalInstancesImp fvars j
1382
1383 partial def withNewLocalInstances {α} (fvars : Array Expr) (j : Nat) : n α → n α :=
1384   mapMetaM <| withNewLocalInstancesImpAux fvars j
1385
1386 @[inline] private def forallTelescopeImp {α} (type : Expr) (k : Array Expr → Expr → MetaM α) : MetaM α := do
1387   forallTelescopeReducingAuxAux (reducing := false) (maxFVars? := none) type k
1388
1389 /--
1390   Given `type` of the form `forall xs, A`, execute `k xs A`.
1391   This combinator will declare local declarations, create free variables for them,
1392   execute `k` with updated local context, and make sure the cache is restored after executing `k`. -/
1393 def forallTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
1394   map2MetaM (fun k => forallTelescopeImp type k) k
1395
1396 private def forallTelescopeReducingImp {α} (type : Expr) (k : Array Expr → Expr → MetaM α) : MetaM α :=
1397   forallTelescopeReducingAux type (maxFVars? := none) k
1398
1399 /--

```

```

1400 Similar to `forallTelescope`, but given `type` of the form `forall xs, A`,
1401 it reduces `A` and continues building the telescope if it is a `forall`. -/
1402 def forallTelescopeReducing {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
1403   map2MetaM (fun k => forallTelescopeReducingImp type k) k
1404
1405 private def forallBoundedTelescopeImp {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → MetaM α) : MetaM α :=
1406   forallTelescopeReducingAux type maxFVars? k
1407
1408 /--
1409 Similar to `forallTelescopeReducing`, stops constructing the telescope when
1410 it reaches size `maxFVars`. -/
1411 def forallBoundedTelescope {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → n α) : n α :=
1412   map2MetaM (fun k => forallBoundedTelescopeImp type maxFVars? k) k
1413
1414 /-- Similar to `forallTelescopeAuxAux` but for lambda and let expressions. -/
1415 private partial def lambdaTelescopeAux {α}
1416   (k : Array Expr → Expr → MetaM α)
1417   : Bool → LocalContext → Array Expr → Nat → Expr → MetaM α
1418 | consumeLet, lctx, fvars, j, Expr.lam n d b c => do
1419   let d := d.instantiateRevRange j fvars.size fvars
1420   let fvarId ← mkFreshId
1421   let lctx := lctx.mkLocalDecl fvarId n d c.binderInfo
1422   let fvar := mkFVar fvarId
1423   lambdaTelescopeAux k consumeLet lctx (fvars.push fvar) j b
1424 | true, lctx, fvars, j, Expr.letE n t v b _ => do
1425   let t := t.instantiateRevRange j fvars.size fvars
1426   let v := v.instantiateRevRange j fvars.size fvars
1427   let fvarId ← mkFreshId
1428   let lctx := lctx.mkLetDecl fvarId n t v
1429   let fvar := mkFVar fvarId
1430   lambdaTelescopeAux k true lctx (fvars.push fvar) j b
1431 | _, lctx, fvars, j, e =>
1432   let e := e.instantiateRevRange j fvars.size fvars;
1433   withReader (fun ctx => { ctx with lctx := lctx }) do
1434     withNewLocalInstancesImp fvars j do
1435       k fvars e
1436
1437 private partial def lambdaTelescopeImp {α} (e : Expr) (consumeLet : Bool) (k : Array Expr → Expr → MetaM α) : MetaM α := do
1438   let rec process (consumeLet : Bool) (lctx : LocalContext) (fvars : Array Expr) (j : Nat) (e : Expr) : MetaM α := do
1439     match consumeLet, e with
1440     | _, Expr.lam n d b c =>
1441       let d := d.instantiateRevRange j fvars.size fvars
1442       let fvarId ← mkFreshId
1443       let lctx := lctx.mkLocalDecl fvarId n d c.binderInfo
1444       let fvar := mkFVar fvarId
1445       process consumeLet lctx (fvars.push fvar) j b
1446     | true, Expr.letE n t v b _ => do

```

```

1447     let t := t.instantiateRevRange j fvars.size fvars
1448     let v := v.instantiateRevRange j fvars.size fvars
1449     let fvarId ← mkFreshId
1450     let lctx := lctx.mkLetDecl fvarId n t v
1451     let fvar := mkFVar fvarId
1452     process true lctx (fvars.push fvar) j b
1453 | _, e =>
1454   let e := e.instantiateRevRange j fvars.size fvars
1455   withReader (fun ctx => { ctx with lctx := lctx }) do
1456     withNewLocalInstancesImp fvars j do
1457       k fvars e
1458 process consumeLet (← getLCtx) #[] 0 e
1459
1460 /-- Similar to `forallTelescope` but for lambda and let expressions. -/
1461 def lambdaLetTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
1462   map2MetaM (fun k => lambdaTelescopeImp type true k) k
1463
1464 /-- Similar to `forallTelescope` but for lambda expressions. -/
1465 def lambdaTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
1466   map2MetaM (fun k => lambdaTelescopeImp type false k) k
1467
1468 /-- Return the parameter names for the given global declaration. -/
1469 def getParamNames (declName : Name) : MetaM (Array Name) := do
1470   let cinfo ← getConstInfo declName
1471   forallTelescopeReducing cinfo.type fun xs _ => do
1472     xs.mapM fun x => do
1473       let localDecl ← getLocalDecl x.fvarId!
1474       pure localDecl.userName
1475
1476 -- `kind` specifies the metavariable kind for metavariables not corresponding to instance implicit `[ ... ]` arguments.
1477 private partial def forallMetaTelescopeReducingAux
1478   (e : Expr) (reducing : Bool) (maxMVars? : Option Nat) (kind : MetavarKind) : MetaM (Array Expr × Array BinderInfo × Expr) :=
1479   let rec process (mvars : Array Expr) (bis : Array BinderInfo) (j : Nat) (type : Expr) : MetaM (Array Expr × Array BinderInfo × Expr)
1480   := do
1481     match type with
1482     | Expr.forallE n d b c =>
1483       let cont : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
1484         let d := d.instantiateRevRange j mvars.size mvars
1485         let k := if c.binderInfo.isInstImplicit then MetavarKind.synthetic else kind
1486         let mvar ← mkFreshExprMVar d k n
1487         let mvars := mvars.push mvar
1488         let bis := bis.push c.binderInfo
1489         process mvars bis j b
1490       match maxMVars? with
1491       | none => cont ()
1492       | some maxMVars =>
1493         if mvars.size < maxMVars then

```



```

1493     cont ()
1494   else
1495     let type := type.instantiateRevRange j mvars.size mvars;
1496     pure (mvars, bis, type)
1497 | _ =>
1498   let type := type.instantiateRevRange j mvars.size mvars;
1499   if reducing then do
1500     let newType ← whnf type;
1501     if newType.isForall then
1502       process mvars bis mvars.size newType
1503     else
1504       pure (mvars, bis, type)
1505   else
1506     pure (mvars, bis, type)
1507 process #[] #[] 0 e
1508
1509 /-- Similar to `forallTelescope`, but creates metavariables instead of free variables. -/
1510 def forallMetaTelescope (e : Expr) (kind := MetavarKind.natural) : MetaM (Array Expr × Array BinderInfo × Expr) :=
1511   forallMetaTelescopeReducingAux e (reducing := false) (maxMVars? := none) kind
1512
1513 /-- Similar to `forallTelescopeReducing`, but creates metavariables instead of free variables. -/
1514 def forallMetaTelescopeReducing (e : Expr) (maxMVars? : Option Nat := none) (kind := MetavarKind.natural) : MetaM (Array Expr × Array
BinderInfo × Expr) :=
1515   forallMetaTelescopeReducingAux e (reducing := true) maxMVars? kind
1516
1517 /-- Similar to `forallMetaTelescopeReducingAux` but for lambda expressions. -/
1518 partial def lambdaMetaTelescope (e : Expr) (maxMVars? : Option Nat := none) : MetaM (Array Expr × Array BinderInfo × Expr) :=
1519   let rec process (mvars : Array Expr) (bis : Array BinderInfo) (j : Nat) (type : Expr) : MetaM (Array Expr × Array BinderInfo × Expr)
:= do
1520     let finalize : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
1521       let type := type.instantiateRevRange j mvars.size mvars
1522       pure (mvars, bis, type)
1523     let cont : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
1524       match type with
1525       | Expr.lam n d b c =>
1526         let d := d.instantiateRevRange j mvars.size mvars
1527         let mvar ← mkFreshExprMVar d
1528         let mvars := mvars.push mvar
1529         let bis := bis.push c.binderInfo
1530         process mvars bis j b
1531       | _ => finalize ()
1532     match maxMVars? with
1533     | none => cont ()
1534     | some maxMVars =>
1535       if mvars.size < maxMVars then
1536         cont ()
1537       else

```

```

1538     finalize ()
1539 process #[] #[] 0 e
1540
1541 private def withNewFVar {α} (fvar fvarType : Expr) (k : Expr → MetaM α) : MetaM α := do
1542   match (← isClass? fvarType) with
1543   | none    => k fvar
1544   | some c => withNewLocalInstance c fvar <| k fvar
1545
1546 private def withLocalDeclImp {α} (n : Name) (bi : BinderInfo) (type : Expr) (k : Expr → MetaM α) : MetaM α := do
1547   let fvarId ← mkFreshId
1548   let ctx ← read
1549   let lctx := ctx.lctx.mkLocalDecl fvarId n type bi
1550   let fvar := mkFVar fvarId
1551   withReader (fun ctx => { ctx with lctx := lctx }) do
1552     withNewFVar fvar type k
1553
1554 def withLocalDecl {α} (name : Name) (bi : BinderInfo) (type : Expr) (k : Expr → n α) : n α :=
1555   map1MetaM (fun k => withLocalDeclImp name bi type k) k
1556
1557 def withLocalDeclD {α} (name : Name) (type : Expr) (k : Expr → n α) : n α :=
1558   withLocalDecl name BinderInfo.default type k
1559
1560 private def withLetDeclImp {α} (n : Name) (type : Expr) (val : Expr) (k : Expr → MetaM α) : MetaM α := do
1561   let fvarId ← mkFreshId
1562   let ctx ← read
1563   let lctx := ctx.lctx.mkLetDecl fvarId n type val
1564   let fvar := mkFVar fvarId
1565   withReader (fun ctx => { ctx with lctx := lctx }) do
1566     withNewFVar fvar type k
1567
1568 def withLetDecl {α} (name : Name) (type : Expr) (val : Expr) (k : Expr → n α) : n α :=
1569   map1MetaM (fun k => withLetDeclImp name type val k) k
1570
1571 private def withExistingLocalDeclsImp {α} (decls : List LocalDecl) (k : MetaM α) : MetaM α := do
1572   let ctx ← read
1573   let numLocalInstances := ctx.localInstances.size
1574   let lctx := decls.foldl1 (fun (lctx : LocalContext) decl => lctx.addDecl decl) ctx.lctx
1575   withReader (fun ctx => { ctx with lctx := lctx }) do
1576     let newLocalInsts ← decls.foldlM
1577       (fun (newLocalInsts : Array LocalInstance) (decl : LocalDecl) => (do {
1578         match (← isClass? decl.type) with
1579         | none    => pure newLocalInsts
1580         | some c => pure <| newLocalInsts.push { className := c, fvar := decl.toExpr } }) : MetaM _)
1581     ctx.localInstances;
1582     if newLocalInsts.size == numLocalInstances then
1583       k
1584     else

```

```

1585     resettingSynthInstanceCache <| withReader (fun ctx => { ctx with localInstances := newLocalInsts }) k
1586
1587 def withExistingLocalDecls {α} (decls : List LocalDecl) : n α → n α :=
1588   mapMetaM <| withExistingLocalDeclsImp decls
1589
1590 private def withNewMCtxDepthImp {α} (x : MetaM α) : MetaM α := do
1591   let s ← get
1592   let savedMCtx := s.mctx
1593   modifyMCtx fun mctx => mctx.incDepth
1594   try x finally setMCtx savedMCtx
1595
1596 /--
1597   Save cache and `MetavarContext`, bump the `MetavarContext` depth, execute `x`,
1598   and restore saved data. -/
1599 def withNewMCtxDepth {α} : n α → n α :=
1600   mapMetaM withNewMCtxDepthImp
1601
1602 private def withLocalContextImp {α} (lctx : LocalContext) (localInsts : LocalInstances) (x : MetaM α) : MetaM α := do
1603   let localInstsCurr ← getLocalInstances
1604   withReader (fun ctx => { ctx with lctx := lctx, localInstances := localInsts }) do
1605     if localInsts == localInstsCurr then
1606       x
1607     else
1608       resettingSynthInstanceCache x
1609
1610 def withLCtx {α} (lctx : LocalContext) (localInsts : LocalInstances) : n α → n α :=
1611   mapMetaM <| withLocalContextImp lctx localInsts
1612
1613 private def withMVarContextImp {α} (mvarId : MVarId) (x : MetaM α) : MetaM α := do
1614   let mvarDecl ← getMVarDecl mvarId
1615   withLocalContextImp mvarDecl.lctx mvarDecl.localInstances x
1616
1617 /--
1618   Execute `x` using the given metavariable `LocalContext` and `LocalInstances`.
1619   The type class resolution cache is flushed when executing `x` if its `LocalInstances` are
1620   different from the current ones. -/
1621 def withMVarContext {α} (mvarId : MVarId) : n α → n α :=
1622   mapMetaM <| withMVarContextImp mvarId
1623
1624 private def withMCtxImp {α} (mctx : MetavarContext) (x : MetaM α) : MetaM α := do
1625   let mctx' ← getMCtx
1626   setMCtx mctx
1627   try x finally setMCtx mctx'
1628
1629 def withMCtx {α} (mctx : MetavarContext) : n α → n α :=
1630   mapMetaM <| withMCtxImp mctx
1631

```

```

1632 @[inline] private def approxDefEqImp {α} (x : MetaM α) : MetaM α :=
1633   withConfig (fun config => { config with foApprox := true, ctxApprox := true, quasiPatternApprox := true }) x
1634
1635 /-- Execute `x` using approximate unification: `foApprox`, `ctxApprox` and `quasiPatternApprox`. -/
1636 @[inline] def approxDefEq {α} : n α → n α :=
1637   mapMetaM approxDefEqImp
1638
1639 @[inline] private def fullApproxDefEqImp {α} (x : MetaM α) : MetaM α :=
1640   withConfig (fun config => { config with foApprox := true, ctxApprox := true, quasiPatternApprox := true, constApprox := true }) x
1641
1642 /--
1643   Similar to `approxDefEq`, but uses all available approximations.
1644   We don't use `constApprox` by default at `approxDefEq` because it often produces undesirable solution for monadic code.
1645   For example, suppose we have `pure (x > 0)` which has type `?m Prop`. We also have the goal `[Pure ?m]`.
1646   Now, assume the expected type is `!0 Bool`. Then, the unification constraint `?m Prop =?= !0 Bool` could be solved
1647   as `?m := fun _ => !0 Bool` using `constApprox`, but this spurious solution would generate a failure when we try to
1648   solve `[Pure (fun _ => !0 Bool)]` -/
1649 @[inline] def fullApproxDefEq {α} : n α → n α :=
1650   mapMetaM fullApproxDefEqImp
1651
1652 def normalizeLevel (u : Level) : MetaM Level := do
1653   let u ← instantiateLevelMVars u
1654   pure u.normalize
1655
1656 def assignLevelMVar (mvarId : MVarId) (u : Level) : MetaM Unit := do
1657   modifyMCtx fun mctx => mctx.assignLevel mvarId u
1658
1659 def whnfR (e : Expr) : MetaM Expr :=
1660   withTransparency TransparencyMode.reducible <| whnf e
1661
1662 def whnfD (e : Expr) : MetaM Expr :=
1663   withTransparency TransparencyMode.default <| whnf e
1664
1665 def whnfI (e : Expr) : MetaM Expr :=
1666   withTransparency TransparencyMode.instances <| whnf e
1667
1668 def setInlineAttribute (declName : Name) (kind := Compiler.InlineAttributeKind.inline) : MetaM Unit := do
1669   let env ← getEnv
1670   match Compiler.setInlineAttribute env declName kind with
1671   | Except.ok env    => setEnv env
1672   | Except.error msg => throwError msg
1673
1674 private partial def instantiateForallAux (ps : Array Expr) (i : Nat) (e : Expr) : MetaM Expr := do
1675   if h : i < ps.size then
1676     let p := ps.get (i, h)
1677     let e ← whnf e
1678     match e with

```

```

1679 | Expr.forallE _ _ b _ => instantiateForallAux ps (i+1) (b.instantiate1 p)
1680 | _ => throwError "invalid instantiateForall, too many parameters"
1681 else
1682   pure e
1683
1684 /- Given `e` of the form `forall (a_1 : A_1) ... (a_n : A_n), B[a_1, ..., a_n]` and `p_1 : A_1, ... p_n : A_n`, return `B[p_1, ...,
p_n]`. -/
1685 def instantiateForall (e : Expr) (ps : Array Expr) : MetaM Expr :=
1686   instantiateForallAux ps 0 e
1687
1688 private partial def instantiateLambdaAux (ps : Array Expr) (i : Nat) (e : Expr) : MetaM Expr := do
1689   if h : i < ps.size then
1690     let p := ps.get (i, h)
1691     let e ← whnf e
1692     match e with
1693     | Expr.lam _ _ b _ => instantiateLambdaAux ps (i+1) (b.instantiate1 p)
1694     | _ => throwError "invalid instantiateLambda, too many parameters"
1695   else
1696     pure e
1697
1698 /- Given `e` of the form `fun (a_1 : A_1) ... (a_n : A_n) => t[a_1, ..., a_n]` and `p_1 : A_1, ... p_n : A_n`, return `t[p_1, ...,
p_n]`. It uses `whnf` to reduce `e` if it is not a lambda -/
1699
1700 def instantiateLambda (e : Expr) (ps : Array Expr) : MetaM Expr :=
1701   instantiateLambdaAux ps 0 e
1702
1703 /- Return true iff `e` depends on the free variable `fvarId` -/
1704 def dependsOn (e : Expr) (fvarId : FVarId) : MetaM Bool :=
1705   return (← getMCtx).exprDependsOn e fvarId
1706
1707 def ppExpr (e : Expr) : MetaM Format := do
1708   let env ← getEnv
1709   let mctx ← getMCtx
1710   let lctx ← getLCtx
1711   let opts ← getOptions
1712   let ctxCore ← readThe Core.Context
1713   Lean.ppExpr { env := env, mctx := mctx, lctx := lctx, opts := opts, currNamespace := ctxCore.currNamespace, openDecls :=
ctxCore.openDecls } e
1714
1715 @[inline] protected def orelse {α} (x y : MetaM α) : MetaM α := do
1716   let env ← getEnv
1717   let mctx ← getMCtx
1718   try x catch _ => setEnv env; setMCtx mctx; y
1719
1720 instance {α} : OrElse (MetaM α) := {Meta.orelse}
1721
1722 @[inline] private def orelseMergeErrorsImp {α} (x y : MetaM α)

```

```

1723     (mergeRef : Syntax → Syntax → Syntax := fun r₁ r₂ => r₁)
1724     (mergeMsg : MessageData → MessageData → MessageData := fun m₁ m₂ => m₁ ++ Format.line ++ m₂) : MetaM α := do
1725   let env ← getEnv
1726   let mctx ← getMCtx
1727   try
1728     x
1729   catch ex =>
1730     setEnv env
1731     setMCtx mctx
1732     match ex with
1733     | Exception.error ref₁ m₁ =>
1734       try
1735         y
1736       catch
1737         | Exception.error ref₂ m₂ => throw <| Exception.error (mergeRef ref₁ ref₂) (mergeMsg m₁ m₂)
1738         | ex => throw ex
1739     | ex => throw ex
1740
1741 /--
1742   Similar to `orelse`, but merge errors. Note that internal errors are not caught.
1743   The default `mergeRef` uses the `ref` (position information) for the first message.
1744   The default `mergeMsg` combines error messages using `Format.line ++ Format.line` as a separator. -/
1745 @[inline] def orelseMergeErrors {α m} [MonadControlT MetaM m] [Monad m] (x y : m α)
1746   (mergeRef : Syntax → Syntax → Syntax := fun r₁ r₂ => r₁)
1747   (mergeMsg : MessageData → MessageData → MessageData := fun m₁ m₂ => m₁ ++ Format.line ++ Format.line ++ m₂) : m α := do
1748   controlAt MetaM fun runInBase => orelseMergeErrorsImp (runInBase x) (runInBase y) mergeRef mergeMsg
1749
1750 /-- Execute `x`, and apply `f` to the produced error message -/
1751 def mapErrorImp {α} (x : MetaM α) (f : MessageData → MessageData) : MetaM α := do
1752   try
1753     x
1754   catch
1755     | Exception.error ref msg => throw <| Exception.error ref <| f msg
1756     | ex => throw ex
1757
1758 @[inline] def mapError {α m} [MonadControlT MetaM m] [Monad m] (x : m α) (f : MessageData → MessageData) : m α :=
1759   controlAt MetaM fun runInBase => mapErrorImp (runInBase x) f
1760
1761 /-- `commitWhenSome? x` executes `x` and keep modifications when it returns `some a`. -/
1762 @[specialize] def commitWhenSome? {α} (x? : MetaM (Option α)) : MetaM (Option α) := do
1763   let env ← getEnv
1764   let mctx ← getMCtx
1765   try
1766     match (← x?) with
1767     | some a => pure (some a)
1768     | none   =>
1769       setEnv env

```

```

1770     setMCtx mctx
1771     pure none
1772   catch ex =>
1773     setEnv env
1774     setMCtx mctx
1775     throw ex
1776
1777 end Methods
1778 end Meta
1779
1780 export Meta (MetaM)
1781
1782 end Lean
1783 // ::::::::::::::
1784 // Check.lean
1785 // ::::::::::::::
1786 /-
1787 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
1788 Released under Apache 2.0 license as described in the file LICENSE.
1789 Authors: Leonardo de Moura
1790 -/
1791 import Lean.Meta.InferType
1792 import Lean.Meta.LevelDefEq
1793
1794 /-
1795 This is not the Kernel type checker, but an auxiliary method for checking
1796 whether terms produced by tactics and `isDefEq` are type correct.
1797 -/
1798
1799 namespace Lean.Meta
1800
1801 private def ensureType (e : Expr) : MetaM Unit := do
1802   discard <| getLevel e
1803
1804 def throwLetTypeMismatchMessage {α} (fvarId : FVarId) : MetaM α := do
1805   let lctx ← getLCtx
1806   match lctx.find? fvarId with
1807   | some (LocalDecl.ldecl _ _ n t v _) => do
1808     let vType ← inferType v
1809     throwError! "invalid let declaration, term{indentExpr v}\nhas type{indentExpr vType}\nbut is expected to have type{indentExpr t}"
1810   | _ => unreachable!
1811
1812 private def checkConstant (constName : Name) (us : List Level) : MetaM Unit := do
1813   let cinfo ← getConstInfo constName
1814   unless us.length == cinfo.levelParams.length do
1815     throwIncorrectNumberOfLevels constName us
1816

```

```

1817 private def getFunctionDomain (f : Expr) : MetaM (Expr × BinderInfo) := do
1818   let fType ← inferType f
1819   let fType ← whnfD fType
1820   match fType with
1821   | Expr.forallE _ d _ c => return (d, c.binderInfo)
1822   | _                    => throwFunctionExpected f
1823
1824 /-
1825 Given to expressions `a` and `b`, this method tries to annotate terms with `pp.explicit := true` to
1826 expose "implicit" differences. For example, suppose `a` and `b` are of the form
1827 ```lean
1828 @HashMap Nat Nat eqInst hasInst1
1829 @HashMap Nat Nat eqInst hasInst2
1830 ```
1831 By default, the pretty printer formats both of them as `HashMap Nat Nat`.
1832 So, counterintuitive error messages such as
1833 ```lean
1834 error: application type mismatch
1835   HashMap.insert m
1836 argument
1837   m
1838 has type
1839   HashMap Nat Nat
1840 but is expected to have type
1841   HashMap Nat Nat
1842 ```
1843 would be produced.
1844 By adding `pp.explicit := true`, we can generate the more informative error
1845 ```lean
1846 error: application type mismatch
1847   HashMap.insert m
1848 argument
1849   m
1850 has type
1851   @HashMap Nat Nat eqInst hasInst1
1852 but is expected to have type
1853   @HashMap Nat Nat eqInst hasInst2
1854 ```
1855 Remark: this method implements a simple heuristic, we should extend it as we find other counterintuitive
1856 error messages.
1857 -/
1858 partial def addPPExplicitToExposeDiff (a b : Expr) : MetaM (Expr × Expr) := do
1859   if (← getOptions).getBool `pp.all false || (← getOptions).getBool `pp.explicit false then
1860     return (a, b)
1861   else
1862     visit a b
1863 where

```



```

1864 visit (a b : Expr) : MetaM (Expr × Expr) := do
1865   try
1866     if !a.isApp || !b.isApp then
1867       return (a, b)
1868     else if a.getAppNumArgs != b.getAppNumArgs then
1869       return (a, b)
1870     else if not (← isDefEq a.getAppFn b.getAppFn) then
1871       return (a, b)
1872     else
1873       let fType ← inferType a.getAppFn
1874       forallBoundedTelescope fType a.getAppNumArgs fun xs _ => do
1875         let mut as := a.getAppArgs
1876         let mut bs := b.getAppArgs
1877         if (← hasExplicitDiff xs as bs) then
1878           return (a, b)
1879         else
1880           for i in [:as.size] do
1881             let (ai, bi) ← visit as[i] bs[i]
1882             as := as.set! i ai
1883             bs := bs.set! i bi
1884           let a := mkAppN a.getAppFn as
1885           let b := mkAppN b.getAppFn bs
1886           return (a.setAppPPExplicit, b.setAppPPExplicit)
1887   catch _ =>
1888     return (a, b)
1889
1890 hasExplicitDiff (xs as bs : Array Expr) : MetaM Bool := do
1891   for i in [:xs.size] do
1892     let localDecl ← getLocalDecl xs[i].fvarId!
1893     if localDecl.binderInfo.isExplicit then
1894       if not (← isDefEq as[i] bs[i]) then
1895         return true
1896   return false
1897
1898 /-
1899   Return error message "has type{givenType}\nbut is expected to have type{expectedType}"
1900 -/
1901 def mkHasTypeButIsExpectedMsg (givenType expectedType : Expr) : MetaM MessageData := do
1902   let (givenType, expectedType) ← addPPExplicitToExposeDiff givenType expectedType
1903   m!"has type{indentExpr givenType}\nbut is expected to have type{indentExpr expectedType}"
1904
1905 def throwAppTypeMismatch {α} (f a : Expr) (extraMsg : MessageData := Format.nil) : MetaM α := do
1906   let (expectedType, binfo) ← getFunctionDomain f
1907   let mut e := mkApp f a
1908   unless binfo.isExplicit do
1909     e := e.setAppPPExplicit
1910   let aType ← inferType a

```

```

1911 throwError! "application type mismatch{indentExpr e}\nargument{indentExpr a}\n{← mkHasTypeButIsExpectedMsg aType expectedType}"
1912
1913 def checkApp (f a : Expr) : MetaM Unit := do
1914   let fType ← inferType f
1915   let fType ← whnf fType
1916   match fType with
1917   | Expr.forallE _ d _ =>
1918     let aType ← inferType a
1919     unless (← isDefEq d aType) do
1920       throwAppTypeMismatch f a
1921   | _ => throwFunctionExpected (mkApp f a)
1922
1923 private partial def checkAux : Expr → MetaM Unit
1924   | e@(Expr.forallE ..) => checkForall e
1925   | e@(Expr.lam ..)     => checkLambdaLet e
1926   | e@(Expr.letE ..)    => checkLambdaLet e
1927   | Expr.const c lvls _ => checkConstant c lvls
1928   | Expr.app f a _      => do checkAux f; checkAux a; checkApp f a
1929   | Expr.mdata _ e _    => checkAux e
1930   | Expr.proj _ _ e _   => checkAux e
1931   | _                   => pure ()
1932 where
1933   checkLambdaLet (e : Expr) : MetaM Unit :=
1934     lambdaLetTelescope e fun xs b => do
1935       xs.forM fun x => do
1936         let xDecl ← getFVarLocalDecl x;
1937         match xDecl with
1938         | LocalDecl.cdecl (type := t) .. =>
1939           ensureType t
1940           checkAux t
1941         | LocalDecl.ldecl (type := t) (value := v) .. =>
1942           ensureType t
1943           checkAux t
1944           let vType ← inferType v
1945           unless (← isDefEq t vType) do throwLetTypeMismatchMessage x.fvarId!
1946           checkAux v
1947       checkAux b
1948
1949   checkForall (e : Expr) : MetaM Unit :=
1950     forallTelescope e fun xs b => do
1951       xs.forM fun x => do
1952         let xDecl ← getFVarLocalDecl x
1953         ensureType xDecl.type
1954         checkAux xDecl.type
1955       ensureType b
1956       checkAux b
1957

```

```

1958 def check (e : Expr) : MetaM Unit :=
1959   traceCtx `Meta.check do
1960     withTransparency TransparencyMode.all $ checkAux e
1961
1962 def isTypeCorrect (e : Expr) : MetaM Bool := do
1963   try
1964     check e
1965     pure true
1966   catch ex =>
1967     trace[Meta.typeError]! ex.toMessageData
1968     pure false
1969
1970 builtin_initialize
1971   registerTraceClass `Meta.check
1972   registerTraceClass `Meta.typeError
1973
1974 end Lean.Meta
1975 // ::::::::::::::
1976 // Closure.lean
1977 // ::::::::::::::
1978 /-
1979 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
1980 Released under Apache 2.0 license as described in the file LICENSE.
1981 Authors: Leonardo de Moura
1982 -/
1983 import Std.ShareCommon
1984 import Lean.MetavarContext
1985 import Lean.Environment
1986 import Lean.Util.FoldConsts
1987 import Lean.Meta.Basic
1988 import Lean.Meta.Check
1989
1990 /-
1991
1992 This module provides functions for "closing" open terms and
1993 creating auxiliary definitions. Here, we say a term is "open" if
1994 it contains free/meta-variables.
1995
1996 The "closure" is performed by lambda abstracting the
1997 free/meta-variables. Recall that in dependent type theory
1998 lambda abstracting a let-variable may produce type incorrect terms.
1999 For example, given the context
2000 ```lean
2001 (n : Nat := 20)
2002 (x : Vector α n)
2003 (y : Vector α 20)
2004 ```

```

```

2005 the term `x = y` is correct. However, its closure using lambda abstractions
2006 is not.
2007 ```lean
2008 fun (n : Nat) (x : Vector α n) (y : Vector α 20) => x = y
2009 ```
2010 A previous version of this module would address this issue by
2011 always use let-expressions to abstract let-vars. In the example above,
2012 it would produce
2013 ```lean
2014 let n : Nat := 20; fun (x : Vector α n) (y : Vector α 20) => x = y
2015 ```
2016 This approach produces correct result, but produces unsatisfactory
2017 results when we want to create auxiliary definitions.
2018 For example, consider the context
2019 ```lean
2020 (x : Nat)
2021 (y : Nat := fact x)
2022 ```
2023 and the term `h (g y)`, now suppose we want to create an auxiliary definition for `y`.
2024 The previous version of this module would compute the auxiliary definition
2025 ```lean
2026 def aux := fun (x : Nat) => let y : Nat := fact x; h (g y)
2027 ```
2028 and would return the term `aux x` as a substitute for `h (g y)`.
2029 This is correct, but we will re-evaluate `fact x` whenever we use `aux`.
2030 In this module, we produce
2031 ```lean
2032 def aux := fun (y : Nat) => h (g y)
2033 ```
2034 Note that in this particular case, it is safe to lambda abstract the let-variable `y`.
2035 This module uses the following approach to decide whether it is safe or not to lambda
2036 abstract a let-variable.
2037 1) We enable zeta-expansion tracking in `MetaM`. That is, whenever we perform type checking
2038     if a let-variable needs to zeta expanded, we store it in the set `zetaFVarIds`.
2039     We say a let-variable is zeta expanded when we replace it with its value.
2040 2) We use the `MetaM` type checker `check` to type check the expression we want to close,
2041     and the type of the binders.
2042 3) If a let-variable is not in `zetaFVarIds`, we lambda abstract it.
2043
2044 Remark: We still use let-expressions for let-variables in `zetaFVarIds`, but we move the
2045 `let` inside the lambdas. The idea is to make sure the auxiliary definition does not have
2046 an interleaving of `lambda` and `let` expressions. Thus, if the let-variable occurs in
2047 the type of one of the lambdas, we simply zeta-expand it there.
2048 As a final example consider the context
2049 ```lean
2050 (x_1 : Nat)
2051 (x_2 : Nat)

```

```

2052 (x_3 : Nat)
2053 (x   : Nat := fact (10 + x_1 + x_2 + x_3))
2054 (ty  : Type := Nat → Nat)
2055 (f    : ty := fun x => x)
2056 (n    : Nat := 20)
2057 (z    : f 10)
2058 ```
2059 and we use this module to compute an auxiliary definition for the term
2060 ```lean
2061 (let y : { v : Nat // v = n } := ⟨20, rfl⟩; y.1 + n + f x, z + 10)
2062 ```
2063 we obtain
2064 ```lean
2065 def aux (x : Nat) (f : Nat → Nat) (z : Nat) : Nat×Nat :=
2066   let n : Nat := 20;
2067   (let y : {v // v=n} := {val := 20, property := ex._proof_1}; y.val+n+f x, z+10)
2068 ```
2069
2070 BTW, this module also provides the `zeta : Bool` flag. When set to true, it
2071 expands all let-variables occurring in the target expression.
2072 -/
2073
2074 namespace Lean.Meta
2075 namespace Closure
2076
2077 structure ToProcessElement where
2078   fvarId : FVarId
2079   newFVarId : FVarId
2080   deriving Inhabited
2081
2082 structure Context where
2083   zeta : Bool
2084
2085 structure State where
2086   visitedLevel      : LevelMap Level := {}
2087   visitedExpr       : ExprStructMap Expr := {}
2088   levelParams       : Array Name := #[]
2089   nextLevelIdx      : Nat := 1
2090   levelArgs         : Array Level := #[]
2091   newLocalDecls     : Array LocalDecl := #[]
2092   newLocalDeclsForMVars : Array LocalDecl := #[]
2093   newLetDecls       : Array LocalDecl := #[]
2094   nextExprIdx       : Nat := 1
2095   exprMVarArgs      : Array Expr := #[]
2096   exprFVarArgs      : Array Expr := #[]
2097   toProcess         : Array ToProcessElement := #[]
2098

```

```

2099 abbrev ClosureM := ReaderT Context $ StateRefT State MetaM
2100
2101 @[inline] def visitLevel (f : Level → ClosureM Level) (u : Level) : ClosureM Level := do
2102   if !u.hasMVar && !u.hasParam then
2103     pure u
2104   else
2105     let s ← get
2106     match s.visitedLevel.find? u with
2107     | some v => pure v
2108     | none   => do
2109       let v ← f u
2110       modify fun s => { s with visitedLevel := s.visitedLevel.insert u v }
2111       pure v
2112
2113 @[inline] def visitExpr (f : Expr → ClosureM Expr) (e : Expr) : ClosureM Expr := do
2114   if !e.hasLevelParam && !e.hasFVar && !e.hasMVar then
2115     pure e
2116   else
2117     let s ← get
2118     match s.visitedExpr.find? e with
2119     | some r => pure r
2120     | none   =>
2121       let r ← f e
2122       modify fun s => { s with visitedExpr := s.visitedExpr.insert e r }
2123       pure r
2124
2125 def mkNewLevelParam (u : Level) : ClosureM Level := do
2126   let s ← get
2127   let p := (u).appendIndexAfter s.nextLevelIdx
2128   modify fun s => { s with levelParams := s.levelParams.push p, nextLevelIdx := s.nextLevelIdx + 1, levelArgs := s.levelArgs.push u }
2129   pure $ mkLevelParam p
2130
2131 partial def collectLevelAux : Level → ClosureM Level
2132 | u@(Level.succ v _) => return u.updateSucc! (← visitLevel collectLevelAux v)
2133 | u@(Level.max v w _) => return u.updateMax! (← visitLevel collectLevelAux v) (← visitLevel collectLevelAux w)
2134 | u@(Level.imax v w _) => return u.updateIMax! (← visitLevel collectLevelAux v) (← visitLevel collectLevelAux w)
2135 | u@(Level.mvar mvarId _) => mkNewLevelParam u
2136 | u@(Level.param _ _) => mkNewLevelParam u
2137 | u@(Level.zero _) => pure u
2138
2139 def collectLevel (u : Level) : ClosureM Level := do
2140   -- u ← instantiateLevelMVars u
2141   visitLevel collectLevelAux u
2142
2143 def preprocess (e : Expr) : ClosureM Expr := do
2144   let e ← instantiateMVars e
2145   let ctx ← read

```

```

2146 -- If we are not zeta-expanding let-decls, then we use `check` to find
2147 -- which let-decls are dependent. We say a let-decl is dependent if its lambda abstraction is type incorrect.
2148 if !ctx.zeta then
2149   check e
2150   pure e
2151
2152 /--
2153   Remark: This method does not guarantee unique user names.
2154   The correctness of the procedure does not rely on unique user names.
2155   Recall that the pretty printer takes care of unintended collisions. -/
2156 def mkNextUserName : ClosureM Name := do
2157   let s ← get
2158   let n := (`_x).appendIndexAfter s.nextExprIdx
2159   modify fun s => { s with nextExprIdx := s.nextExprIdx + 1 }
2160   pure n
2161
2162 def pushToProcess (elem : ToProcessElement) : ClosureM Unit :=
2163   modify fun s => { s with toProcess := s.toProcess.push elem }
2164
2165 partial def collectExprAux (e : Expr) : ClosureM Expr := do
2166   let collect (e : Expr) := visitExpr collectExprAux e
2167   match e with
2168   | Expr.proj _ _ s _ => return e.updateProj! (← collect s)
2169   | Expr.forallE _ d b _ => return e.updateForallE! (← collect d) (← collect b)
2170   | Expr.lam _ d b _ => return e.updateLambdaE! (← collect d) (← collect b)
2171   | Expr.letE _ t v b _ => return e.updateLet! (← collect t) (← collect v) (← collect b)
2172   | Expr.app f a _ => return e.updateApp! (← collect f) (← collect a)
2173   | Expr.mdata _ b _ => return e.updateMData! (← collect b)
2174   | Expr.sort u _ => return e.updateSort! (← collectLevel u)
2175   | Expr.const c us _ => return e.updateConst! (← us.mapM collectLevel)
2176   | Expr.mvar mvarId _ =>
2177     let mvarDecl ← getMVarDecl mvarId
2178     let type ← preprocess mvarDecl.type
2179     let type ← collect type
2180     let newFVarId ← mkFreshFVarId
2181     let userName ← mkNextUserName
2182     modify fun s => { s with
2183       newLocalDeclsForMVars := s.newLocalDeclsForMVars.push $ LocalDecl.cdecl arbitrary newFVarId userName type BinderInfo.default,
2184       exprMVarArgs          := s.exprMVarArgs.push e
2185     }
2186     return mkFVar newFVarId
2187   | Expr.fvar fvarId _ =>
2188     match (← read).zeta, (← getLocalDecl fvarId).value? with
2189     | true, some value => collect (← preprocess value)
2190     | _, _ =>
2191       let newFVarId ← mkFreshFVarId
2192       pushToProcess (fvarId, newFVarId)

```

```

2193     return mkFVar newFVarId
2194 | e => pure e
2195
2196 def collectExpr (e : Expr) : ClosureM Expr := do
2197   let e ← preprocess e
2198   visitExpr collectExprAux e
2199
2200 partial def pickNextToProcessAux (lctx : LocalContext) (i : Nat) (toProcess : Array ToProcessElement) (elem : ToProcessElement)
2201   : ToProcessElement × Array ToProcessElement :=
2202   if h : i < toProcess.size then
2203     let elem' := toProcess.get {i, h}
2204     if (lctx.get! elem.fvarId).index < (lctx.get! elem'.fvarId).index then
2205       pickNextToProcessAux lctx (i+1) (toProcess.set {i, h} elem) elem'
2206     else
2207       pickNextToProcessAux lctx (i+1) toProcess elem
2208   else
2209     (elem, toProcess)
2210
2211 def pickNextToProcess? : ClosureM (Option ToProcessElement) := do
2212   let lctx ← getLCtx
2213   let s ← get
2214   if s.toProcess.isEmpty then
2215     pure none
2216   else
2217     modifyGet fun s =>
2218       let elem      := s.toProcess.back
2219       let toProcess := s.toProcess.pop
2220       let (elem, toProcess) := pickNextToProcessAux lctx 0 toProcess elem
2221       (some elem, { s with toProcess := toProcess })
2222
2223 def pushFVarArg (e : Expr) : ClosureM Unit :=
2224   modify fun s => { s with exprFVarArgs := s.exprFVarArgs.push e }
2225
2226 def pushLocalDecl (newFVarId : FVarId) (userName : Name) (type : Expr) (bi := BinderInfo.default) : ClosureM Unit := do
2227   let type ← collectExpr type
2228   modify fun s => { s with newLocalDecls := s.newLocalDecls.push <| LocalDecl.cdecl arbitrary newFVarId userName type bi }
2229
2230 partial def process : ClosureM Unit := do
2231   match (← pickNextToProcess?) with
2232   | none => pure ()
2233   | some (fvarId, newFVarId) =>
2234     let localDecl ← getLocalDecl fvarId
2235     match localDecl with
2236     | LocalDecl.cdecl _ _ userName type bi =>
2237       pushLocalDecl newFVarId userName type bi
2238       pushFVarArg (mkFVar fvarId)
2239     process

```



```

2240 | LocalDecl.ldecl _ _ userName type val _ =>
2241   let zetaFVarIds ← getZetaFVarIds
2242   if !zetaFVarIds.contains fvarId then
2243     /- Non-dependent let-decl
2244
2245     Recall that if `fvarId` is in `zetaFVarIds`, then we zeta-expanded it
2246     during type checking (see `check` at `collectExpr`).
2247
2248     Our type checker may zeta-expand declarations that are not needed, but this
2249     check is conservative, and seems to work well in practice. -/
2250     pushLocalDecl newFVarId userName type
2251     pushFVarArg (mkFVar fvarId)
2252     process
2253   else
2254     /- Dependent let-decl -/
2255     let type ← collectExpr type
2256     let val ← collectExpr val
2257     modify fun s => { s with newLetDecls := s.newLetDecls.push <| LocalDecl.ldecl arbitrary newFVarId userName type val false }
2258     /- We don't want to interleave let and lambda declarations in our closure. So, we expand any occurrences of newFVarId
2259     at `newLocalDecls` -/
2260     modify fun s => { s with newLocalDecls := s.newLocalDecls.map (replaceFVarIdAtLocalDecl newFVarId val) }
2261     process
2262
2263 @[inline] def mkBinding (isLambda : Bool) (decls : Array LocalDecl) (b : Expr) : Expr :=
2264   let xs := decls.map LocalDecl.toExpr
2265   let b := b.abstract xs
2266   decls.size.foldRev (init := b) fun i b =>
2267     let decl := decls[i]
2268     match decl with
2269     | LocalDecl.cdecl _ _ n ty bi =>
2270       let ty := ty.abstractRange i xs
2271       if isLambda then
2272         Lean.mkLambda n bi ty b
2273       else
2274         Lean.mkForall n bi ty b
2275     | LocalDecl.ldecl _ _ n ty val nonDep =>
2276       if b.hasLooseBVar 0 then
2277         let ty := ty.abstractRange i xs
2278         let val := val.abstractRange i xs
2279         mkLet n ty val b nonDep
2280       else
2281         b.lowerLooseBVars 1 1
2282
2283 def mkLambda (decls : Array LocalDecl) (b : Expr) : Expr :=
2284   mkBinding true decls b
2285
2286 def mkForall (decls : Array LocalDecl) (b : Expr) : Expr :=

```

```

2287 mkBinding false decls b
2288
2289 structure MkValueTypeClosureResult where
2290   levelParams : Array Name
2291   type         : Expr
2292   value        : Expr
2293   levelArgs    : Array Level
2294   exprArgs     : Array Expr
2295
2296 def mkValueTypeClosureAux (type : Expr) (value : Expr) : ClosureM (Expr × Expr) := do
2297   resetZetaFVarIds
2298   withTrackingZeta do
2299     let type ← collectExpr type
2300     let value ← collectExpr value
2301     process
2302     pure (type, value)
2303
2304 def mkValueTypeClosure (type : Expr) (value : Expr) (zeta : Bool) : MetaM MkValueTypeClosureResult := do
2305   let ((type, value), s) ← ((mkValueTypeClosureAux type value).run { zeta := zeta }).run {}
2306   let newLocalDecls := s.newLocalDecls.reverse ++ s.newLocalDeclsForMVars
2307   let newLetDecls   := s.newLetDecls.reverse
2308   let type := mkForall newLocalDecls (mkForall newLetDecls type)
2309   let value := mkLambda newLocalDecls (mkLambda newLetDecls value)
2310   pure {
2311     type      := type,
2312     value     := value,
2313     levelParams := s.levelParams,
2314     levelArgs  := s.levelArgs,
2315     exprArgs   := s.exprFVarArgs.reverse ++ s.exprMVarArgs
2316   }
2317
2318 end Closure
2319
2320 /--
2321   Create an auxiliary definition with the given name, type and value.
2322   The parameters `type` and `value` may contain free and meta variables.
2323   A "closure" is computed, and a term of the form `name.{u_1 ... u_n} t_1 ... t_m` is
2324   returned where `u_i`'s are universe parameters and metavariables `type` and `value` depend on,
2325   and `t_j`'s are free and meta variables `type` and `value` depend on. -/
2326 def mkAuxDefinition (name : Name) (type : Expr) (value : Expr) (zeta : Bool := false) (compile : Bool := true) : MetaM Expr := do
2327   trace[Meta.debug]! "{name} : {type} := {value}"
2328   let result ← Closure.mkValueTypeClosure type value zeta
2329   let env ← getEnv
2330   let decl := Declaration.defnDecl {
2331     name      := name,
2332     levelParams := result.levelParams.toList,
2333     type      := result.type,

```

```

2334     value      := result.value,
2335     hints      := ReducibilityHints.regular (getMaxHeight env result.value + 1),
2336     safety     := if env.hasUnsafe result.type || env.hasUnsafe result.value then DefinitionSafety.unsafe else DefinitionSafety.safe
2337   }
2338   trace[Meta.debug]! "{name} : {result.type} := {result.value}"
2339   addDecl decl
2340   if compile then
2341     compileDecl decl
2342   return mkAppN (mkConst name result.levelArgs.toList) result.exprArgs
2343
2344 /-- Similar to `mkAuxDefinition`, but infers the type of `value`. -/
2345 def mkAuxDefinitionFor (name : Name) (value : Expr) : MetaM Expr := do
2346   let type ← inferType value
2347   let type := type.headBeta
2348   mkAuxDefinition name type value
2349
2350 end Lean.Meta
2351 // ::::::::::::::
2352 // Coe.lean
2353 // ::::::::::::::
2354 /-
2355 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
2356 Released under Apache 2.0 license as described in the file LICENSE.
2357 Authors: Leonardo de Moura
2358 -/
2359 import Lean.Meta.WHNF
2360 import Lean.Meta.Transform
2361
2362 namespace Lean.Meta
2363
2364 /--
2365   Return true iff `declName` is one of the auxiliary definitions/projections
2366   used to implement coercions.
2367 -/
2368 def isCoeDecl (declName : Name) : Bool :=
2369   declName == ``coe ||
2370   declName == ``coeB || declName == ``coeHead || declName == ``coeTail || declName == ``coeD ||
2371   declName == ``coeTC || declName == ``coeFun || declName == ``coeSort ||
2372   declName == ``Coe.coe || declName == ``CoeTC.coe || declName == ``CoeHead.coe ||
2373   declName == ``CoeTail.coe || declName == ``CoeHTCT.coe || declName == ``CoeDep.coe ||
2374   declName == ``CoeT.coe || declName == ``CoeFun.coe || declName == ``CoeSort.coe ||
2375   declName == ``liftCoeM || declName == ``coeM
2376
2377 /-- Expand coercions occurring in `e` -/
2378 partial def expandCoe (e : Expr) : MetaM Expr :=
2379   withReducibleAndInstances do
2380     return (← transform e (pre := step))

```

```

2381 where
2382   step (e : Expr) : MetaM TransformStep := do
2383     let f := e.getAppFn
2384     if !f.isConst then
2385       return TransformStep.visit e
2386     else
2387       let declName := f.constName!
2388       if isCoeDecl declName then
2389         match (← unfoldDefinition? e) with
2390         | none   => return TransformStep.visit e
2391         | some e => step e.headBeta
2392       else
2393         return TransformStep.visit e
2394
2395 end Lean.Meta
2396 // ::::::::::::::
2397 // CollectMVars.lean
2398 // ::::::::::::::
2399 /-
2400 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
2401 Released under Apache 2.0 license as described in the file LICENSE.
2402 Authors: Leonardo de Moura
2403 -/
2404 import Lean.Util.CollectMVars
2405 import Lean.Meta.Basic
2406
2407 namespace Lean.Meta
2408
2409 /--
2410   Collect unassigned metavariables occurring in the given expression.
2411
2412   Remark: if `e` contains `?m` and there is a `t` assigned to `?m`, we
2413   collect unassigned metavariables occurring in `t`.
2414
2415   Remark: if `e` contains `?m` and `?m` is delayed assigned to some term `t`,
2416   we collect `?m` and unassigned metavariables occurring in `t`.
2417   We collect `?m` because it has not been assigned yet. -/
2418 partial def collectMVars (e : Expr) : StateRefT CollectMVars.State MetaM Unit := do
2419   let e ← instantiateMVars e
2420   let s ← get
2421   let resultSavedSize := s.result.size
2422   let s := e.collectMVars s
2423   set s
2424   for mvarId in s.result[resultSavedSize:] do
2425     match (← getDelayedAssignment? mvarId) with
2426     | none   => pure ()
2427     | some d => collectMVars d.val

```

```

2428
2429 /-- Return metavariables in occurring the given expression. See `collectMVars` -/
2430 def getMVars (e : Expr) : MetaM (Array MVarId) := do
2431   let (_, s) ← (collectMVars e).run {}
2432   pure s.result
2433
2434 /-- Similar to getMVars, but removes delayed assignments. -/
2435 def getMVarsNoDelayed (e : Expr) : MetaM (Array MVarId) := do
2436   let mvarIds ← getMVars e
2437   mvarIds.filterM fun mvarId => not <$> isDelayedAssigned mvarId
2438
2439 def collectMVarsAtDecl (d : Declaration) : StateRefT CollectMVars.State MetaM Unit :=
2440   d.forExprM collectMVars
2441
2442 def getMVarsAtDecl (d : Declaration) : MetaM (Array MVarId) := do
2443   let (_, s) ← (collectMVarsAtDecl d).run {}
2444   pure s.result
2445
2446 end Lean.Meta
2447 // ::::::::::::::
2448 // DiscrTree.lean
2449 // ::::::::::::::
2450 /-
2451 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
2452 Released under Apache 2.0 license as described in the file LICENSE.
2453 Authors: Leonardo de Moura
2454 -/
2455 import Lean.Meta.Basic
2456 import Lean.Meta.FunInfo
2457 import Lean.Meta.InferType
2458
2459 namespace Lean.Meta.DiscrTree
2460 /-
2461   (Imperfect) discrimination trees.
2462   We use a hybrid representation.
2463   - A `PersistentHashMap` for the root node which usually contains many children.
2464   - A sorted array of key/node pairs for inner nodes.
2465
2466   The edges are labeled by keys:
2467   - Constant names (and arity). Universe levels are ignored.
2468   - Free variables (and arity). Thus, an entry in the discrimination tree
2469     may reference hypotheses from the local context.
2470   - Literals
2471   - Star/Wildcard. We use them to represent metavariables and terms
2472     we want to ignore. We ignore implicit arguments and proofs.
2473   - Other. We use to represent other kinds of terms (e.g., nested lambda, forall, sort, etc).
2474

```

```

2475 We reduce terms using `TransparencyMode.reducible`. Thus, all reducible
2476 definitions in an expression `e` are unfolded before we insert it into the
2477 discrimination tree.
2478
2479 Recall that projections from classes are **NOT** reducible.
2480 For example, the expressions `Add.add α (ringAdd ?α ?s) ?x ?x`
2481 and `Add.add Nat Nat.hasAdd a b` generates paths with the following keys
2482 respectively
2483 ```
2484 {Add.add, 4}, *, *, *, *
2485 {Add.add, 4}, *, *, {a,0}, {b,0}
2486 ```
2487
2488 That is, we don't reduce `Add.add Nat inst a b` into `Nat.add a b`.
2489 We say the `Add.add` applications are the de-facto canonical forms in
2490 the metaprogramming framework.
2491 Moreover, it is the metaprogrammer's responsibility to re-pack applications such as
2492 `Nat.add a b` into `Add.add Nat inst a b`.
2493
2494 Remark: we store the arity in the keys
2495 1- To be able to implement the "skip" operation when retrieving "candidate"
2496    unifiers.
2497 2- Distinguish partial applications `f a`, `f a b`, and `f a b c`.
2498 -/
2499
2500 def Key.ctorIdx : Key → Nat
2501 | Key.star      => 0
2502 | Key.other     => 1
2503 | Key.lit _     => 2
2504 | Key.fvar _ _  => 3
2505 | Key.const _ _ => 4
2506
2507 def Key.lt : Key → Key → Bool
2508 | Key.lit v₁,      Key.lit v₂      => v₁ < v₂
2509 | Key.fvar n₁ a₁,  Key.fvar n₂ a₂  => Name.quickLt n₁ n₂ || (n₁ == n₂ && a₁ < a₂)
2510 | Key.const n₁ a₁, Key.const n₂ a₂ => Name.quickLt n₁ n₂ || (n₁ == n₂ && a₁ < a₂)
2511 | k₁,             k₂               => k₁.ctorIdx < k₂.ctorIdx
2512
2513 instance : HasLess Key := {fun a b => Key.lt a b}
2514 instance (a b : Key) : Decidable (a < b) := inferInstanceAs (Decidable (Key.lt a b))
2515
2516 def Key.format : Key → Format
2517 | Key.star      => "*"
2518 | Key.other     => "■"
2519 | Key.lit (Literal.natVal v) => fmt v
2520 | Key.lit (Literal.strVal v) => repr v
2521 | Key.const k _      => fmt k

```

```

2522 | Key.fvar k _          => fmt k
2523
2524 instance : ToFormat Key := (Key.format)
2525
2526 def Key.arity : Key → Nat
2527 | Key.const _ a => a
2528 | Key.fvar _ a  => a
2529 | _             => 0
2530
2531 instance {α} : Inhabited (Trie α) := (Trie.node #[] #[])
2532
2533 def empty {α} : DiscrTree α := { root := {} }
2534
2535 partial def Trie.format {α} [ToFormat α] : Trie α → Format
2536 | Trie.node vs cs => Format.group $ Format.paren $
2537   "node" ++ (if vs.isEmpty then Format.nil else " " ++ fmt vs)
2538   ++ Format.join (cs.toList.map $ fun (k, c) => Format.line ++ Format.paren (fmt k ++ " => " ++ format c))
2539
2540 instance {α} [ToFormat α] : ToFormat (Trie α) := (Trie.format)
2541
2542 partial def format {α} [ToFormat α] (d : DiscrTree α) : Format :=
2543   let (_, r) := d.root.foldl
2544     (fun (p : Bool × Format) k c =>
2545       (false, p.2 ++ (if p.1 then Format.nil else Format.line) ++ Format.paren (fmt k ++ " => " ++ fmt c)))
2546     (true, Format.nil)
2547   Format.group r
2548
2549 instance {α} [ToFormat α] : ToFormat (DiscrTree α) := (format)
2550
2551 /- The discrimination tree ignores implicit arguments and proofs.
2552    We use the following auxiliary id as a "mark". -/
2553 private def tmpMVarId : MVarId := `discr_tree_tmp
2554 private def tmpStar  := mkMVar tmpMVarId
2555
2556 instance {α} : Inhabited (DiscrTree α) where
2557   default := {}
2558
2559 /--
2560   Return true iff the argument should be treated as a "wildcard" by the discrimination tree.
2561
2562   - We ignore proofs because of proof irrelevance. It doesn't make sense to try to
2563     index their structure.
2564
2565   - We ignore instance implicit arguments (e.g., `[Add α]`) because they are "morally" canonical.
2566     Moreover, we may have many definitionally equal terms floating around.
2567     Example: `Ring.hasAdd Int Int.isRing` and `Int.hasAdd`.
2568

```

```

2569 - We considered ignoring implicit arguments (e.g., `{α : Type}`) since users don't "see" them,
2570 and may not even understand why some simplification rule is not firing.
2571 However, in type class resolution, we have instance such as `Decidable (@Eq Nat x y)`,
2572 where `Nat` is an implicit argument. Thus, we would add the path
2573 ```
2574 Decidable -> Eq -> * -> * -> * -> [Nat.decEq]
2575 ```
2576 to the discrimination tree IF we ignored the implicit `Nat` argument.
2577 This would be BAD since **ALL** decidable equality instances would be in the same path.
2578 So, we index implicit arguments if they are types.
2579 This setting seems sensible for simplification lemmas such as:
2580 ```
2581 forall (x y : Unit), (@Eq Unit x y) = true
2582 ```
2583 If we ignore the implicit argument `Unit`, the `DiscrTree` will say it is a candidate
2584 simplification lemma for any equality in our goal.
2585
2586 Remark: if users have problems with the solution above, we may provide a `noIndexing` annotation,
2587 and `ignoreArg` would return true for any term of the form `noIndexing t`.
2588 -/
2589 private def ignoreArg (a : Expr) (i : Nat) (infos : Array ParamInfo) : MetaM Bool :=
2590   if h : i < infos.size then
2591     let info := infos.get {i, h}
2592     if info.instImplicit then
2593       pure true
2594     else if info.implicit then
2595       not <$> isType a
2596     else
2597       isProof a
2598   else
2599     isProof a
2600
2601 private partial def pushArgsAux (infos : Array ParamInfo) : Nat → Expr → Array Expr → MetaM (Array Expr)
2602   | i, Expr.app f a _, todo => do
2603     if (← ignoreArg a i infos) then
2604       pushArgsAux infos (i-1) f (todo.push tmpStar)
2605     else
2606       pushArgsAux infos (i-1) f (todo.push a)
2607   | _, _, todo => pure todo
2608
2609 private partial def whnfEta (e : Expr) : MetaM Expr := do
2610   let e ← whnf e
2611   match e.etaExpandedStrict? with
2612   | some e => whnfEta e
2613   | none   => pure e
2614
2615 /-

```



```

2616 TODO: add a parameter (wildcardConsts : NameSet) to `DiscrTree.insert`.
2617 Then, `DiscrTree` users may control which symbols should be treated as wildcards.
2618 Different `DiscrTree` users may populate this set using, for example, attributes.
2619
2620 Remark: we currently tag `Nat.zero` and `Nat.succ` to avoid having to add special
2621 support for `Expr.lit`. Example, suppose the discrimination tree contains the entry
2622 `Nat.succ ?m |-> v`, and we are trying to retrieve the matches for `Expr.lit (Literal.natVal 1) _`.
2623 In this scenario, we want to retrieve `Nat.succ ?m |-> v` -/
2624 private def shouldAddAsStar (constName : Name) : Bool :=
2625   constName == `Nat.zero || constName == `Nat.succ
2626
2627 def mkNoindexAnnotation (e : Expr) : Expr :=
2628   mkAnnotation `noindex e
2629
2630 def hasNoindexAnnotation (e : Expr) : Bool :=
2631   annotation? `noindex e |>.isSome
2632
2633 /- Remark: we use `shouldAddAsStar` only for nested terms, and `first == false` for nested terms -/
2634
2635 private def pushArgs (first : Bool) (todo : Array Expr) (e : Expr) : MetaM (Key × Array Expr) := do
2636   if hasNoindexAnnotation e then
2637     return (Key.star, todo)
2638   else
2639     let e ← whnfEta e
2640     let fn := e.getAppFn
2641     let push (k : Key) (nargs : Nat) : MetaM (Key × Array Expr) := do
2642       let info ← getFunInfoNArgs fn nargs
2643       let todo ← pushArgsAux info.paramInfo (nargs-1) e todo
2644       return (k, todo)
2645     match fn with
2646     | Expr.lit v _ => return (Key.lit v, todo)
2647     | Expr.const c _ =>
2648       if !first && shouldAddAsStar c then
2649         return (Key.star, todo)
2650       else
2651         let nargs := e.getAppNumArgs
2652         push (Key.const c nargs) nargs
2653     | Expr.fvar fvarId _ =>
2654       let nargs := e.getAppNumArgs
2655       push (Key.fvar fvarId nargs) nargs
2656     | Expr.mvar mvarId _ =>
2657       if mvarId == tmpMVarId then
2658         -- We use `tmp` to mark implicit arguments and proofs
2659         return (Key.star, todo)
2660       else if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
2661         return (Key.other, todo)
2662     else

```

```

2663     return (Key.star, todo)
2664 | _ =>
2665     return (Key.other, todo)
2666
2667 partial def mkPathAux (first : Bool) (todo : Array Expr) (keys : Array Key) : MetaM (Array Key) := do
2668   if todo.isEmpty then
2669     pure keys
2670   else
2671     let e := todo.back
2672     let todo := todo.pop
2673     let (k, todo) ← pushArgs first todo e
2674     mkPathAux false todo (keys.push k)
2675
2676 private def initCapacity := 8
2677
2678 def mkPath (e : Expr) : MetaM (Array Key) := do
2679   withReducible do
2680     let todo : Array Expr := Array.mkEmpty initCapacity
2681     let keys : Array Key := Array.mkEmpty initCapacity
2682     mkPathAux (first := true) (todo.push e) keys
2683
2684 private partial def createNodes {α} (keys : Array Key) (v : α) (i : Nat) : Trie α :=
2685   if h : i < keys.size then
2686     let k := keys.get {i, h}
2687     let c := createNodes keys v (i+1)
2688     Trie.node #[] #[(k, c)]
2689   else
2690     Trie.node #[v] #[]
2691
2692 private def insertVal {α} [BEq α] (vs : Array α) (v : α) : Array α :=
2693   if vs.contains v then vs else vs.push v
2694
2695 private partial def insertAux {α} [BEq α] (keys : Array Key) (v : α) : Nat → Trie α → Trie α
2696 | i, Trie.node vs cs =>
2697   if h : i < keys.size then
2698     let k := keys.get {i, h}
2699     let c := Id.run $ cs.binInsertM
2700       (fun a b => a.1 < b.1)
2701       (fun (_, s) => let c := insertAux keys v (i+1) s; (k, c)) -- merge with existing
2702       (fun _ => let c := createNodes keys v (i+1); (k, c))
2703       (k, arbitrary)
2704     Trie.node vs c
2705   else
2706     Trie.node (insertVal vs v) cs
2707
2708 def insertCore {α} [BEq α] (d : DiscrTree α) (keys : Array Key) (v : α) : DiscrTree α :=
2709   if keys.isEmpty then panic! "invalid key sequence"

```

```

2710 else
2711   let k := keys[0]
2712   match d.root.find? k with
2713   | none =>
2714     let c := createNodes keys v 1
2715     { root := d.root.insert k c }
2716   | some c =>
2717     let c := insertAux keys v 1 c
2718     { root := d.root.insert k c }
2719
2720 def insert {α} [BEq α] (d : DiscrTree α) (e : Expr) (v : α) : MetaM (DiscrTree α) := do
2721   let keys ← mkPath e
2722   return d.insertCore keys v
2723
2724 private def getKeyArgs (e : Expr) (isMatch? : Bool) : MetaM (Key × Array Expr) := do
2725   let e ← whnfEta e
2726   match e.getAppFn with
2727   | Expr.lit v _ => pure (Key.lit v, #[])
2728   | Expr.const c _ _ =>
2729     let nargs := e.getAppNumArgs
2730     pure (Key.const c nargs, e.getAppRevArgs)
2731   | Expr.fvar fvarId _ =>
2732     let nargs := e.getAppNumArgs
2733     pure (Key.fvar fvarId nargs, e.getAppRevArgs)
2734   | Expr.mvar mvarId _ =>
2735     if isMatch? then
2736       pure (Key.other, #[])
2737     else do
2738       let ctx ← read
2739       if ctx.config.isDefEqStuckEx then
2740         /-
2741           When the configuration flag `isDefEqStuckEx` is set to true,
2742           we want `isDefEq` to throw an exception whenever it tries to assign
2743           a read-only metavariable.
2744           This feature is useful for type class resolution where
2745           we may want to notify the caller that the TC problem may be solveable
2746           later after it assigns `?m`.
2747           The method `DiscrTree.getUnify e` returns candidates `c` that may "unify" with `e`.
2748           That is, `isDefEq c e` may return true. Now, consider `DiscrTree.getUnify d (Add ?m)`
2749           where `?m` is a read-only metavariable, and the discrimination tree contains the keys
2750           `HadAdd Nat` and `Add Int`. If `isDefEqStuckEx` is set to true, we must treat `?m` as
2751           a regular metavariable here, otherwise we return the empty set of candidates.
2752           This is incorrect because it is equivalent to saying that there is no solution even if
2753           the caller assigns `?m` and try again. -/
2754         pure (Key.star, #[])
2755       else if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
2756         pure (Key.other, #[])

```

```

2757     else
2758         pure (Key.star, #[])
2759 | _ => pure (Key.other, #[])
2760
2761 private abbrev getMatchKeyArgs (e : Expr) : MetaM (Key × Array Expr) :=
2762   getKeyArgs e true
2763
2764 private abbrev getUnifyKeyArgs (e : Expr) : MetaM (Key × Array Expr) :=
2765   getKeyArgs e false
2766
2767 private def getStarResult {α} (d : DiscrTree α) : Array α :=
2768   let result : Array α := Array.mkEmpty initCapacity
2769   match d.root.find? Key.star with
2770   | none      => result
2771   | some (Trie.node vs _) => result ++ vs
2772
2773 partial def getMatch {α} (d : DiscrTree α) (e : Expr) : MetaM (Array α) :=
2774   withReducible do
2775     let result := getStarResult d
2776     let (k, args) ← getMatchKeyArgs e
2777     match k with
2778     | Key.star => pure result
2779     | _       =>
2780       match d.root.find? k with
2781       | none  => pure result
2782       | some c => process args c result
2783 where
2784   process (todo : Array Expr) (c : Trie α) (result : Array α) : MetaM (Array α) := do
2785     match c with
2786     | Trie.node vs cs =>
2787       if todo.isEmpty then
2788         return result ++ vs
2789       else if cs.isEmpty then
2790         return result
2791     else
2792       let e      := todo.back
2793       let todo   := todo.pop
2794       let first := cs[0] /- Recall that `Key.star` is the minimal key -/
2795       let (k, args) ← getMatchKeyArgs e
2796       /- We must always visit `Key.star` edges since they are wildcards.
2797          Thus, `todo` is not used linearly when there is `Key.star` edge
2798          and there is an edge for `k` and `k != Key.star`. -/
2799       let visitStarChild (result : Array α) : MetaM (Array α) :=
2800         if first.1 == Key.star then
2801           process todo first.2 result
2802         else
2803           return result

```



```

2851         | some c => process 0 (todo ++ args) c.2 (← visitStarChild result)
2852
2853 end Lean.Meta.DiscrTree
2854 // ::::::::::::::
2855 // DiscrTreeTypes.lean
2856 // ::::::::::::::
2857 /-
2858 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
2859 Released under Apache 2.0 license as described in the file LICENSE.
2860 Authors: Leonardo de Moura
2861 -/
2862 import Lean.Expr
2863
2864 namespace Lean.Meta
2865
2866 /- See file `DiscrTree.lean` for the actual implementation and documentation. -/
2867
2868 namespace DiscrTree
2869
2870 inductive Key where
2871   | const : Name → Nat → Key
2872   | fvar  : FVarId → Nat → Key
2873   | lit   : Literal → Key
2874   | star  : Key
2875   | other : Key
2876   deriving Inhabited, BEq
2877
2878 protected def Key.hash : Key → USize
2879   | Key.const n a => mixHash 5237 $ mixHash (hash n) (hash a)
2880   | Key.fvar n a  => mixHash 3541 $ mixHash (hash n) (hash a)
2881   | Key.lit v     => mixHash 1879 $ hash v
2882   | Key.star      => 7883
2883   | Key.other     => 2411
2884
2885 instance : Hashable Key := {Key.hash}
2886
2887 inductive Trie (α : Type) where
2888   | node (vs : Array α) (children : Array (Key × Trie α)) : Trie α
2889
2890 end DiscrTree
2891
2892 open DiscrTree
2893 open Std (PersistentHashMap)
2894
2895 structure DiscrTree (α : Type) where
2896   root : PersistentHashMap Key (Trie α) := {}
2897

```

```

2898 end Lean.Meta
2899 // ::::::::::::::
2900 // ExprDefEq.lean
2901 // ::::::::::::::
2902 /-
2903 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
2904 Released under Apache 2.0 license as described in the file LICENSE.
2905 Authors: Leonardo de Moura
2906 -/
2907 import Lean.ProjFns
2908 import Lean.Meta.WHNF
2909 import Lean.Meta.InferType
2910 import Lean.Meta.FunInfo
2911 import Lean.Meta.LevelDefEq
2912 import Lean.Meta.Check
2913 import Lean.Meta.Offset
2914 import Lean.Meta.ForEachExpr
2915 import Lean.Meta.UnificationHint
2916
2917 namespace Lean.Meta
2918
2919 /--
2920   Try to solve `a := (fun x => t) ==?= b` by eta-expanding `b`.
2921
2922   Remark: eta-reduction is not a good alternative even in a system without universe cumulativity like Lean.
2923   Example:
2924     ```
2925     (fun x : A => f ?m) ==?= f
2926     ```
2927     The left-hand side of the constraint above it not eta-reduced because `?m` is a metavariable. -/
2928 private def isDefEqEta (a b : Expr) : MetaM Bool := do
2929   if a.isLambda && !b.isLambda then
2930     let bType ← inferType b
2931     let bType ← whnfD bType
2932     match bType with
2933     | Expr.forallE n d _ c =>
2934       let b' := mkLambda n c.binderInfo d (mkApp b (mkBVar 0))
2935       commitWhen <| Meta.isExprDefEqAux a b'
2936     | _ => pure false
2937   else
2938     pure false
2939
2940 /-- Support for `Lean.reduceBool` and `Lean.reduceNat` -/
2941 def isDefEqNative (s t : Expr) : MetaM LBool := do
2942   let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
2943   let s? ← reduceNative? s
2944   let t? ← reduceNative? t

```

```

2945 match s?, t? with
2946 | some s, some t => isDefEq s t
2947 | some s, none   => isDefEq s t
2948 | none,   some t => isDefEq s t
2949 | none,   none   => pure LBool.undef
2950
2951 /-- Support for reducing Nat basic operations. -/
2952 def isDefEqNat (s t : Expr) : MetaM LBool := do
2953   let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
2954   if s.hasFVar || s.hasMVar || t.hasFVar || t.hasMVar then
2955     pure LBool.undef
2956   else
2957     let s? ← reduceNat? s
2958     let t? ← reduceNat? t
2959     match s?, t? with
2960     | some s, some t => isDefEq s t
2961     | some s, none   => isDefEq s t
2962     | none,   some t => isDefEq s t
2963     | none,   none   => pure LBool.undef
2964
2965 /-- Support for constraints of the form `("..." == String.mk cs)` -/
2966 def isDefEqStringLit (s t : Expr) : MetaM LBool := do
2967   let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
2968   if s.isStringLit && t.isAppOf `String.mk then
2969     isDefEq (toCtorIfLit s) t
2970   else if s.isAppOf `String.mk && t.isStringLit then
2971     isDefEq s (toCtorIfLit t)
2972   else
2973     pure LBool.undef
2974
2975 /--
2976   Return `true` if `e` is of the form `fun (x_1 ... x_n) => ?m x_1 ... x_n`, and `?m` is unassigned.
2977   Remark: `n` may be 0. -/
2978 def isEtaUnassignedMVar (e : Expr) : MetaM Bool := do
2979   match e.etaExpanded? with
2980   | some (Expr.mvar mvarId _) =>
2981     if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
2982       pure false
2983     else if (← isExprMVarAssigned mvarId) then
2984       pure false
2985     else
2986       pure true
2987   | _ => pure false
2988
2989 /--
2990   First pass for `isDefEqArgs`. We unify explicit arguments, *and* easy cases
2991   Here, we say a case is easy if it is of the form

```



```

2992
2993     ?m ==?= t
2994     or
2995     t ==?= ?m
2996
2997 where `?m` is unassigned.
2998
2999 These easy cases are not just an optimization. When
3000 `?m` is a function, by assigning it to t, we make sure
3001 a unification constraint (in the explicit part)
3002 ```
3003 ?m t ==?= f s
3004 ```
3005 is not higher-order.
3006
3007 We also handle the eta-expanded cases:
3008 ```
3009 fun x₁ ... xₙ => ?m x₁ ... xₙ ==?= t
3010 t ==?= fun x₁ ... xₙ => ?m x₁ ... xₙ
3011
3012 This is important because type inference often produces
3013 eta-expanded terms, and without this extra case, we could
3014 introduce counter intuitive behavior.
3015
3016 Pre: `paramInfo.size <= args₁.size = args₂.size`
3017 -/
3018 private partial def isDefEqArgsFirstPass
3019   (paramInfo : Array ParamInfo) (args₁ args₂ : Array Expr) : MetaM (Option (Array Nat)) := do
3020   let rec loop (i : Nat) (postponed : Array Nat) := do
3021     if h : i < paramInfo.size then
3022       let info := paramInfo.get (i, h)
3023       let a₁ := args₁[i]
3024       let a₂ := args₂[i]
3025       if info.implicit || info.instImplicit then
3026         if (← isEtaUnassignedMVar a₁ <||> isEtaUnassignedMVar a₂) then
3027           if (← Meta.isExprDefEqAux a₁ a₂) then
3028             loop (i+1) postponed
3029           else
3030             pure none
3031         else
3032           loop (i+1) (postponed.push i)
3033       else if (← Meta.isExprDefEqAux a₁ a₂) then
3034         loop (i+1) postponed
3035       else
3036         pure none
3037     else
3038       pure (some postponed)

```

```

3039 loop 0 #[]
3040
3041 @[specialize] private def trySynthPending (e : Expr) : MetaM Bool := do
3042   let mvarId? ← getStuckMVar? e
3043   match mvarId? with
3044   | some mvarId => Meta.synthPending mvarId
3045   | none       => pure false
3046
3047 private partial def isDefEqArgs (f : Expr) (args1 args2 : Array Expr) : MetaM Bool :=
3048   if h : args1.size = args2.size then do
3049     let finfo ← getFunInfoNArgs f args1.size
3050     let (some postponed) ← isDefEqArgsFirstPass finfo.paramInfo args1 args2 | pure false
3051     let rec processOtherArgs (i : Nat) : MetaM Bool := do
3052       if h1 : i < args1.size then
3053         let a1 := args1.get (i, h1)
3054         let a2 := args2.get (i, Eq.subst h h1)
3055         if (← Meta.isExprDefEqAux a1 a2) then
3056           processOtherArgs (i+1)
3057         else
3058           pure false
3059     else
3060       pure true
3061   if (← processOtherArgs finfo.paramInfo.size) then
3062     postponed.allM fun i => do
3063       /- Second pass: unify implicit arguments.
3064       In the second pass, we make sure we are unfolding at
3065       least non reducible definitions (default setting). -/
3066       let a1 := args1[i]
3067       let a2 := args2[i]
3068       let info := finfo.paramInfo[i]
3069       if info.instImplicit then
3070         discard <| trySynthPending a1
3071         discard <| trySynthPending a2
3072       withAtLeastTransparency TransparencyMode.default <| Meta.isExprDefEqAux a1 a2
3073   else
3074     pure false
3075 else
3076   pure false
3077
3078 /--
3079 Check whether the types of the free variables at `fvars` are
3080 definitionally equal to the types at `ds2`.
3081
3082 Pre: `fvars.size == ds2.size`
3083
3084 This method also updates the set of local instances, and invokes
3085 the continuation `k` with the updated set.

```

```

3086
3087 We can't use `withNewLocalInstances` because the `isDeq fvarType d₂`
3088 may use local instances. -/
3089 @[specialize] partial def isDefEqBindingDomain (fvars : Array Expr) (ds₂ : Array Expr) (k : MetaM Bool) : MetaM Bool :=
3090   let rec loop (i : Nat) := do
3091     if h : i < fvars.size then do
3092       let fvar := fvars.get (i, h)
3093       let fvarDecl ← getFVarLocalDecl fvar
3094       let fvarType := fvarDecl.type
3095       let d₂      := ds₂[i]
3096       if (← Meta.isExprDefEqAux fvarType d₂) then
3097         match (← isClass? fvarType) with
3098         | some className => withNewLocalInstance className fvar <| loop (i+1)
3099         | none           => loop (i+1)
3100       else
3101         pure false
3102     else
3103       k
3104   loop 0
3105
3106 /- Auxiliary function for `isDefEqBinding` for handling binders `forall/fun`.
3107 It accumulates the new free variables in `fvars`, and declare them at `lctx`.
3108 We use the domain types of `e₁` to create the new free variables.
3109 We store the domain types of `e₂` at `ds₂`. -/
3110 private partial def isDefEqBindingAux (lctx : LocalContext) (fvars : Array Expr) (e₁ e₂ : Expr) (ds₂ : Array Expr) : MetaM Bool :=
3111   let process (n : Name) (d₁ d₂ b₁ b₂ : Expr) : MetaM Bool := do
3112     let d₁      := d₁.instantiateRev fvars
3113     let d₂      := d₂.instantiateRev fvars
3114     let fvarId ← mkFreshId
3115     let lctx   := lctx.mkLocalDecl fvarId n d₁
3116     let fvars  := fvars.push (mkFVar fvarId)
3117     isDefEqBindingAux lctx fvars b₁ b₂ (ds₂.push d₂)
3118   match e₁, e₂ with
3119   | Expr.forallE n d₁ b₁ _, Expr.forallE _ d₂ b₂ _ => process n d₁ d₂ b₁ b₂
3120   | Expr.lam      n d₁ b₁ _, Expr.lam      _ d₂ b₂ _ => process n d₁ d₂ b₁ b₂
3121   | _, _         =>
3122     withReader (fun ctx => { _ ctx with lctx := lctx }) do
3123       isDefEqBindingDomain fvars ds₂ do
3124         Meta.isExprDefEqAux (e₁.instantiateRev fvars) (e₂.instantiateRev fvars)
3125
3126 @[inline] private def isDefEqBinding (a b : Expr) : MetaM Bool := do
3127   let lctx ← getLCtx
3128   isDefEqBindingAux lctx #[] a b #[]
3129
3130 private def checkTypesAndAssign (mvar : Expr) (v : Expr) : MetaM Bool :=
3131   traceCtx `Meta.isDefEq.assign.checkTypes do
3132     if !mvar.isMVar then

```

```

3133     trace[Meta.isDefEq.assign.final]! "metavariable expected at {mvar} := {v}"
3134     return false
3135   else
3136     -- must check whether types are definitionally equal or not, before assigning and returning true
3137     let mvarType ← inferType mvar
3138     let vType ← inferType v
3139     if (← withTransparency TransparencyMode.default <| Meta.isExprDefEqAux mvarType vType) then
3140       trace[Meta.isDefEq.assign.final]! "{mvar} := {v}"
3141       assignExprMVar mvar.mvarId! v
3142       pure true
3143     else
3144       trace[Meta.isDefEq.assign.typeMismatch]! "{mvar} : {mvarType} := {v} : {vType}"
3145       pure false
3146
3147 /--
3148 Auxiliary method for solving constraints of the form `?m xs := v`.
3149 It creates a lambda using `mkLambdaFVars ys v`, where `ys` is a superset of `xs`.
3150 `ys` is often equal to `xs`. It is a bigger when there are let-declaration dependencies in `xs`.
3151 For example, suppose we have `xs` of the form `#[a, c]` where
3152 ```
3153 a : Nat
3154 b : Nat := f a
3155 c : b = a
3156 ```
3157 In this scenario, the type of `?m` is `(x1 : Nat) -> (x2 : f x1 = x1) -> C[x1, x2]`,
3158 and type of `v` is `C[a, c]`. Note that, `?m a c` is type correct since `f a = a` is definitionally equal
3159 to the type of `c : b = a`, and the type of `?m a c` is equal to the type of `v`.
3160 Note that `fun xs => v` is the term `fun (x1 : Nat) (x2 : b = x1) => v` which has type
3161 `(x1 : Nat) -> (x2 : b = x1) -> C[x1, x2]` which is not definitionally equal to the type of `?m`,
3162 and may not even be type correct.
3163 The issue here is that we are not capturing the `let`-declarations.
3164
3165 This method collects let-declarations `y` occurring between `xs[0]` and `xs.back` s.t.
3166 some `x` in `xs` depends on `y`.
3167 `ys` is the `xs` with these extra let-declarations included.
3168
3169 In the example above, `ys` is `#[a, b, c]`, and `mkLambdaFVars ys v` produces
3170 `fun a => let b := f a; fun (c : b = a) => v` which has a type definitionally equal to the type of `?m`.
3171
3172 Recall that the method `checkAssignment` ensures `v` does not contain offending `let`-declarations.
3173
3174 This method assumes that for any `xs[i]` and `xs[j]` where `i < j`, we have that `index of xs[i] < index of xs[j]`.
3175 where the index is the position in the local context.
3176 -/
3177 private partial def mkLambdaFVarsWithLetDeps (xs : Array Expr) (v : Expr) : MetaM (Option Expr) := do
3178   if not (← hasLetDeclsInBetween) then
3179     mkLambdaFVars xs v

```

```

3180 else
3181   let ys ← addLetDeps
3182   trace[Meta.debug]! "ys: {ys}, v: {v}"
3183   mkLambdaFVars ys v
3184
3185 where
3186   /- Return true if there are let-declarations between `xs[0]` and `xs[xs.size-1]`.
3187   We use it a quick-check to avoid the more expensive collection procedure. -/
3188   hasLetDeclsInBetween : MetaM Bool := do
3189     let check (lctx : LocalContext) : Bool := do
3190       let start := lctx.getFVar! xs[0] |>.index
3191       let stop  := lctx.getFVar! xs.back |>.index
3192       for i in [start+1:stop] do
3193         match lctx.getAt! i with
3194         | some localDecl =>
3195           if localDecl.isLet then
3196             return true
3197         | _ => pure ()
3198       return false
3199     if xs.size <= 1 then
3200       pure false
3201     else
3202       check (← getLCtx)
3203
3204   /- Traverse `e` and stores in the state `NameHashSet` any let-declaration with index greater than `(← read)`.
3205   The context `Nat` is the position of `xs[0]` in the local context. -/
3206   collectLetDeclsFrom (e : Expr) : ReaderT Nat (StateRefT NameHashSet MetaM) Unit := do
3207     let rec visit (e : Expr) : MonadCacheT Expr Unit (ReaderT Nat (StateRefT NameHashSet MetaM)) Unit :=
3208       checkCache e fun _ => do
3209         match e with
3210         | Expr.forallE _ d b _ => visit d; visit b
3211         | Expr.lam _ d b _    => visit d; visit b
3212         | Expr.letE _ t v b _ => visit t; visit v; visit b
3213         | Expr.app f a _      => visit f; visit a
3214         | Expr.mdata _ b _    => visit b
3215         | Expr.proj _ _ b _   => visit b
3216         | Expr.fvar fvarId _  =>
3217           let localDecl ← getLocalDecl fvarId
3218           if localDecl.isLet && localDecl.index > (← read) then
3219             modify fun s => s.insert localDecl.fvarId
3220         | _ => pure ()
3221     visit (← instantiateMVars e) |>.run
3222
3223   /-
3224   Auxiliary definition for traversing all declarations between `xs[0]` ... `xs.back` backwards.
3225   The `Nat` argument is the current position in the local context being visited, and it is less than
3226   or equal to the position of `xs.back` in the local context.

```

```

3227   The `Nat` context `(← read)` is the position of `xs[0]` in the local context.
3228   -/
3229   collectLetDepsAux : Nat → ReaderT Nat (StateRefT NameHashSet MetaM) Unit
3230   | 0 => return ()
3231   | i+1 => do
3232     if i+1 == (← read) then
3233       return ()
3234     else
3235       match (← getLCtx).getAt! (i+1) with
3236       | none => collectLetDepsAux i
3237       | some localDecl =>
3238         if (← get).contains localDecl.fvarId then
3239           collectLetDeclsFrom localDecl.type
3240         match localDecl.value? with
3241         | some val => collectLetDeclsFrom val
3242         | _ => pure ()
3243       collectLetDepsAux i
3244
3245   /- Computes the set `ys`. It is a set of `FVarId`s, -/
3246   collectLetDeps : MetaM NameHashSet := do
3247     let lctx ← getLCtx
3248     let start := lctx.getFVar! xs[0] |>.index
3249     let stop  := lctx.getFVar! xs.back |>.index
3250     let s := xs.foldl (init := {}) fun s x => s.insert x.fvarId!
3251     let (_, s) ← collectLetDepsAux stop |>.run start |>.run s
3252     return s
3253
3254   /- Computes the array `ys` containing let-decls between `xs[0]` and `xs.back` that
3255     some `x` in `xs` depends on. -/
3256   addLetDeps : MetaM (Array Expr) := do
3257     let lctx ← getLCtx
3258     let s ← collectLetDeps
3259     /- Convert `s` into the the array `ys` -/
3260     let start := lctx.getFVar! xs[0] |>.index
3261     let stop  := lctx.getFVar! xs.back |>.index
3262     let mut ys := #[]
3263     for i in [start:stop+1] do
3264       match lctx.getAt! i with
3265       | none => pure ()
3266       | some localDecl =>
3267         if s.contains localDecl.fvarId then
3268           ys := ys.push localDecl.toExpr
3269     return ys
3270
3271   /-
3272   Each metavariable is declared in a particular local context.
3273   We use the notation `C |- ?m : t` to denote a metavariable `?m` that

```

```

3274 was declared at the local context `C` with type `t` (see `MetavarDecl`).
3275 We also use `?m@C` as a shorthand for `C |- ?m : t` where `t` is the type of `?m`.
3276
3277 The following method process the unification constraint
3278
3279     ?m@C a1 ... an =?= t
3280
3281 We say the unification constraint is a pattern IFF
3282
3283     1) `a1 ... an` are pairwise distinct free variables that are <200b>*not*<200b> let-variables.
3284     2) `a1 ... an` are not in `C`
3285     3) `t` only contains free variables in `C` and/or `{a1, ..., an}`
3286     4) For every metavariable `?m'@C` occurring in `t`, `C` is a subprefix of `C`
3287     5) `?m` does not occur in `t`
3288
3289 Claim: we don't have to check free variable declarations. That is,
3290 if `t` contains a reference to `x : A := v`, we don't need to check `v`.
3291 Reason: The reference to `x` is a free variable, and it must be in `C` (by 1 and 3).
3292 If `x` is in `C`, then any metavariable occurring in `v` must have been defined in a strict subprefix of `C`.
3293 So, condition 4 and 5 are satisfied.
3294
3295 If the conditions above have been satisfied, then the
3296 solution for the unification constrain is
3297
3298     ?m := fun a1 ... an => t
3299
3300 Now, we consider some workarounds/approximations.
3301
3302 A1) Suppose `t` contains a reference to `x : A := v` and `x` is not in `C` (failed condition 3)
3303     (precise) solution: unfold `x` in `t`.
3304
3305 A2) Suppose some `ai` is in `C` (failed condition 2)
3306     (approximated) solution (when `config.ctxApprox` is set to true) :
3307     ignore condition and also use
3308
3309     ?m := fun a1 ... an => t
3310
3311 Here is an example where this approximation fails:
3312 Given `C` containing `a : nat`, consider the following two constraints
3313     ?m@C a =?= a
3314     ?m@C b =?= a
3315
3316 If we use the approximation in the first constraint, we get
3317     ?m := fun x => x
3318 when we apply this solution to the second one we get a failure.
3319
3320 IMPORTANT: When applying this approximation we need to make sure the

```

```

3321 abstracted term `fun a1 ... an => t` is type correct. The check
3322 can only be skipped in the pattern case described above. Consider
3323 the following example. Given the local context
3324     (α : Type) (a : α)
3325
3326 we try to solve
3327
3328     ?m α =?= @id α a
3329
3330 If we use the approximation above we obtain:
3331
3332     ?m := (fun α' => @id α' a)
3333
3334 which is a type incorrect term. `a` has type `α` but it is expected to have
3335 type `α'`.
3336
3337 The problem occurs because the right hand side contains a free variable
3338 `a` that depends on the free variable `α` being abstracted. Note that
3339 this dependency cannot occur in patterns.
3340
3341 We can address this by type checking
3342 the term after abstraction. This is not a significant performance
3343 bottleneck because this case doesn't happen very often in practice
3344 (262 times when compiling stdlib on Jan 2018). The second example
3345 is trickier, but it also occurs less frequently (8 times when compiling
3346 stdlib on Jan 2018, and all occurrences were at Init/Control when
3347 we define monads and auxiliary combinators for them).
3348 We considered three options for the addressing the issue on the second example:
3349
3350 A3) `a1 ... an` are not pairwise distinct (failed condition 1).
3351 In Lean3, we would try to approximate this case using an approach similar to A2.
3352 However, this approximation complicates the code, and is never used in the
3353 Lean3 stdlib and mathlib.
3354
3355 A4) `t` contains a metavariable `?m@C` where `C` is not a subprefix of `C`.
3356 If `?m` is assigned, we substitute.
3357 If not, we create an auxiliary metavariable with a smaller scope.
3358 Actually, we let `elimMVarDeps` at `MetavarContext.lean` to perform this step.
3359
3360 A5) If some `ai` is not a free variable,
3361 then we use first-order unification (if `config.foApprox` is set to true)
3362
3363     ?m a1 ... ai a{i+1} ... a{i+k} =?= f b1 ... bk
3364
3365 reduces to
3366
3367

```



```

3368      ?M a_1 ... a_i =?= f
3369      a_{i+1}      =?= b_1
3370      ...
3371      a_{i+k}      =?= b_k
3372
3373
3374 A6) If (m =?= v) is of the form
3375
3376      ?m a_1 ... a_n =?= ?m b_1 ... b_k
3377
3378      then we use first-order unification (if `config.foApprox` is set to true)
3379
3380 A7) When `foApprox`, we may use another approximation (`constApprox`) for solving constraints of the form
3381 ```
3382      ?m s_1 ... s_n =?= t
3383      ```
3384      where `s_1 ... s_n` are arbitrary terms. We solve them by assigning the constant function to `?m`.
3385      ```
3386      ?m := fun _ ... _ => t
3387      ```
3388
3389      In general, this approximation may produce bad solutions, and may prevent coercions from being tried.
3390      For example, consider the term `pure (x > 0)` with inferred type `?m Prop` and expected type `IO Bool`.
3391      In this situation, the
3392      elaborator generates the unification constraint
3393      ```
3394      ?m Prop =?= IO Bool
3395      ```
3396      It is not a higher-order pattern, nor first-order approximation is applicable. However, constant approximation
3397      produces the bogus solution `?m := fun _ => IO Bool`, and prevents the system from using the coercion from
3398      the decidable proposition `x > 0` to `Bool`.
3399
3400      On the other hand, the constant approximation is desirable for elaborating the term
3401      ```
3402      let f (x : _) := pure "hello"; f ()
3403      ```
3404      with expected type `IO String`.
3405      In this example, the following unification constraint is generated.
3406      ```
3407      ?m () String =?= IO String
3408      ```
3409      It is not a higher-order pattern, first-order approximation reduces it to
3410      ```
3411      ?m () =?= IO
3412      ```
3413      which fails to be solved. However, constant approximation solves it by assigning
3414      ```

```

```

3415     ?m := fun _ => IO
3416     ``
3417     Note that `f`'s type is `(x : ?α) -> ?m x String`. The metavariable `?m` may depend on `x`.
3418     If `constApprox` is set to true, we use constant approximation. Otherwise, we use a heuristic to decide
3419     whether we should apply it or not. The heuristic is based on observing where the constraints above come from.
3420     In the first example, the constraint `?m Prop == IO Bool` come from polymorphic method where `?m` is expected to
3421     be a function of type `Type -> Type`. In the second example, the first argument of `?m` is used to model
3422     a potential dependency on `x`. By using constant approximation here, we are just saying the type of `f`
3423     does not depend on `x`. We claim this is a reasonable approximation in practice. Moreover, it is expected
3424     by any functional programmer used to non-dependently type languages (e.g., Haskell).
3425     We distinguish the two cases above by using the field `numScopeArgs` at `MetavarDecl`. This field tracks
3426     how many metavariable arguments are representing dependencies.
3427 -/
3428
3429 def mkAuxMVar (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr) (numScopeArgs : Nat := 0) : MetaM Expr := do
3430   mkFreshExprMVarAt lctx localInsts type MetavarKind.natural Name.anonymous numScopeArgs
3431
3432 namespace CheckAssignment
3433
3434 builtin_initialize checkAssignmentExceptionId : InternalExceptionId ← registerInternalExceptionId `checkAssignment
3435 builtin_initialize outOfScopeExceptionId : InternalExceptionId ← registerInternalExceptionId `outOfScope
3436
3437 structure State where
3438   cache : ExprStructMap Expr := {}
3439
3440 structure Context where
3441   mvarId      : MVarId
3442   mvarDecl    : MetavarDecl
3443   fvars       : Array Expr
3444   hasCtxLocals : Bool
3445   rhs         : Expr
3446
3447 abbrev CheckAssignmentM := ReaderT Context $ StateRefT State MetaM
3448
3449 def throwCheckAssignmentFailure {α} : CheckAssignmentM α :=
3450   throw <| Exception.internal checkAssignmentExceptionId
3451
3452 def throwOutOfScopeFVar {α} : CheckAssignmentM α :=
3453   throw <| Exception.internal outOfScopeExceptionId
3454
3455 private def findCached? (e : Expr) : CheckAssignmentM (Option Expr) := do
3456   return (← get).cache.find? e
3457
3458 private def cache (e r : Expr) : CheckAssignmentM Unit := do
3459   modify fun s => { s with cache := s.cache.insert e r }
3460
3461 instance : MonadCache Expr Expr CheckAssignmentM where

```

```

3462 findCached? := findCached?
3463 cache       := cache
3464
3465 @[inline] private def visit (f : Expr → CheckAssignmentM Expr) (e : Expr) : CheckAssignmentM Expr :=
3466   if !e.hasExprMVar && !e.hasFVar then pure e else checkCache e (fun _ => f e)
3467
3468 private def addAssignmentInfo (msg : MessageData) : CheckAssignmentM MessageData := do
3469   let ctx ← read
3470   return m! "{msg} @ {mkMVar ctx.mvarId} {ctx.fvars} := {ctx.rhs}"
3471
3472 @[inline] def run (x : CheckAssignmentM Expr) (mvarId : MVarId) (fvars : Array Expr) (hasCtxLocals : Bool) (v : Expr) : MetaM (Option Expr) := do
3473   let mvarDecl ← getMVarDecl mvarId
3474   let ctx := { mvarId := mvarId, mvarDecl := mvarDecl, fvars := fvars, hasCtxLocals := hasCtxLocals, rhs := v : Context }
3475   let x : CheckAssignmentM (Option Expr) :=
3476     catchInternalIds [outOfScopeExceptionId, checkAssignmentExceptionId]
3477       (do let e ← x; return some e)
3478       (fun _ => pure none)
3479   x.run ctx |>.run' {}
3480
3481 mutual
3482
3483 partial def checkFVar (fvar : Expr) : CheckAssignmentM Expr := do
3484   let ctxMeta ← readThe Meta.Context
3485   let ctx ← read
3486   if ctx.mvarDecl.lctx.containsFVar fvar then
3487     pure fvar
3488   else
3489     let lctx := ctxMeta.lctx
3490     match lctx.findFVar? fvar with
3491     | some (LocalDecl.ldecl (value := v) ..) => visit check v
3492     | _ =>
3493       if ctx.fvars.contains fvar then pure fvar
3494       else
3495         traceM `Meta.isDefEq.assign.outOfScopeFVar do addAssignmentInfo fvar
3496         throwOutOfScopeFVar
3497
3498 partial def checkMVar (mvar : Expr) : CheckAssignmentM Expr := do
3499   let mvarId := mvar.mvarId!
3500   let ctx ← read
3501   let mctx ← getMCtx
3502   if mvarId == ctx.mvarId then
3503     traceM `Meta.isDefEq.assign.occursCheck <| addAssignmentInfo "occurs check failed"
3504     throwCheckAssignmentFailure
3505   else match mctx.getExprAssignment? mvarId with
3506   | some v => check v
3507   | none =>

```

```

3508 match mctx.findDecl? mvarId with
3509 | none      => throwUnknownMVar mvarId
3510 | some mvarDecl =>
3511   if ctx.hasCtxLocals then
3512     throwCheckAssignmentFailure -- It is not a pattern, then we fail and fall back to F0 unification
3513   else if mvarDecl.lctx.isSubPrefixOf ctx.mvarDecl.lctx ctx.fvars then
3514     /- The local context of `mvar` - free variables being abstracted is a subprefix of the metavariable being assigned.
3515     We "subtract" variables being abstracted because we use `elimMVarDeps` -/
3516     pure mvar
3517   else if mvarDecl.depth != mctx.depth || mvarDecl.kind.isSyntheticOpaque then
3518     traceM `Meta.isDefEq.assign.readOnlyMVarWithBiggerLCtx <| addAssignmentInfo (mkMVar mvarId)
3519     throwCheckAssignmentFailure
3520   else
3521     let ctxMeta ← readThe Meta.Context
3522     if ctxMeta.config.ctxApprox && ctx.mvarDecl.lctx.isSubPrefixOf mvarDecl.lctx then
3523       /- Create an auxiliary metavariable with a smaller context and "checked" type.
3524       Note that `mvarType` may be different from `mvarDecl.type`. Example: `mvarType` contains
3525       a metavariable that we also need to reduce the context.
3526
3527       We remove from `ctx.mvarDecl.lctx` any variable that is not in `mvarDecl.lctx`
3528       or in `ctx.fvars`. We don't need to remove the ones in `ctx.fvars` because
3529       `elimMVarDeps` will take care of them.
3530
3531       First, we collect `toErase` the variables that need to be erased.
3532       Notat that if a variable is `ctx.fvars`, but it depends on variable at `toErase`,
3533       we must also erase it.
3534     -/
3535     let toErase := mvarDecl.lctx.foldl (init := #[]) fun toErase localDecl =>
3536       if ctx.mvarDecl.lctx.contains localDecl.fvarId then
3537         toErase
3538       else if ctx.fvars.any fun fvar => fvar.fvarId! == localDecl.fvarId then
3539         if mctx.findLocalDeclDependsOn localDecl fun fvarId => toErase.contains fvarId then
3540           -- localDecl depends on a variable that will be erased. So, we must add it to `toErase` too
3541           toErase.push localDecl.fvarId
3542         else
3543           toErase
3544       else
3545         toErase.push localDecl.fvarId
3546     let lctx := toErase.foldl (init := mvarDecl.lctx) fun lctx toEraseFVar =>
3547       lctx.erase toEraseFVar
3548     /- Compute new set of local instances. -/
3549     let localInsts := mvarDecl.localInstances.filter fun localInst => toErase.contains localInst.fvar.fvarId!
3550     let mvarType ← check mvarDecl.type
3551     let newMVar ← mkAuxMVar lctx localInsts mvarType mvarDecl.numScopeArgs
3552     modifyThe Meta.State fun s => { s with mctx := s.mctx.assignExpr mvarId newMVar }
3553     pure newMVar
3554   else

```

```

3555         traceM `Meta.isDefEq.assign.readOnlyMVarWithBiggerLCtx <| addAssignmentInfo (mkMVar mvarId)
3556         throwCheckAssignmentFailure
3557
3558 /-
3559   Auxiliary function used to "fix" subterms of the form `?m x_1 ... x_n` where `x_i`'s are free variables,
3560   and one of them is out-of-scope.
3561   See `Expr.app` case at `check`.
3562   If `ctxApprox` is true, then we solve this case by creating a fresh metavariable ?n with the correct scope,
3563   an assigning `?m := fun _ ... _ => ?n` -/
3564 partial def assignToConstFun (mvar : Expr) (numArgs : Nat) (newMVar : Expr) : MetaM Bool := do
3565   let mvarType ← inferType mvar
3566   forallBoundedTelescope mvarType numArgs fun xs _ => do
3567     if xs.size != numArgs then pure false
3568     else
3569       let some v ← mkLambdaFVarsWithLetDeps xs newMVar | return false
3570       match (← checkAssignmentAux mvar.mvarId! #[] false v) with
3571       | some v => checkTypesAndAssign mvar v
3572       | none   => return false
3573
3574 -- See checkAssignment
3575 partial def checkAssignmentAux (mvarId : MVarId) (fvars : Array Expr) (hasCtxLocals : Bool) (v : Expr) : MetaM (Option Expr) := do
3576   run (check v) mvarId fvars hasCtxLocals v
3577
3578 partial def check (e : Expr) : CheckAssignmentM Expr := do
3579   match e with
3580   | Expr.mdata _ b _      => return e.updateMData! (← visit check b)
3581   | Expr.proj _ _ s _    => return e.updateProj! (← visit check s)
3582   | Expr.lam _ d b _     => return e.updateLambdaE! (← visit check d) (← visit check b)
3583   | Expr.forallE _ d b _ => return e.updateForallE! (← visit check d) (← visit check b)
3584   | Expr.letE _ t v b _  => return e.updateLet! (← visit check t) (← visit check v) (← visit check b)
3585   | Expr.bvar ..        => return e
3586   | Expr.sort ..        => return e
3587   | Expr.const ..       => return e
3588   | Expr.lit ..         => return e
3589   | Expr.fvar ..        => visit checkFVar e
3590   | Expr.mvar ..        => visit checkMVar e
3591   | Expr.app ..         => e.withApp fun f args => do
3592     let ctxMeta ← readThe Meta.Context
3593     if f.isMVar && ctxMeta.config.ctxApprox && args.all Expr.isFVar then
3594       let f ← visit checkMVar f
3595       catchInternalId outOfScopeExceptionId
3596       (do
3597         let args ← args.mapM (visit check)
3598         return mkAppN f args)
3599     (fun ex => do
3600       if !f.isMVar then
3601         throw ex

```

```

3602         else if (← isDelayedAssigned f.mvarId!) then
3603             throw ex
3604         else
3605             let eType ← inferType e
3606             let mvarType ← check eType
3607             /- Create an auxiliary metavariable with a smaller context and "checked" type, assign `?f := fun _ => ?newMVar`
3608                Note that `mvarType` may be different from `eType`. -/
3609             let ctx ← read
3610             let newMVar ← mkAuxMVar ctx.mvarDecl.lctx ctx.mvarDecl.localInstances mvarType
3611             if (← assignToConstFun f args.size newMVar) then
3612                 pure newMVar
3613             else
3614                 throw ex)
3615     else
3616         let f ← visit check f
3617         let args ← args.mapM (visit check)
3618         return mkAppN f args
3619
3620 end
3621
3622 end CheckAssignment
3623
3624 namespace CheckAssignmentQuick
3625
3626 partial def check
3627     (hasCtxLocals ctxApprox : Bool)
3628     (mctx : MetavarContext) (lctx : LocalContext) (mvarDecl : MetavarDecl) (mvarId : MVarId) (fvars : Array Expr) (e : Expr) : Bool :=
3629     let rec visit (e : Expr) : Bool :=
3630         if !e.hasExprMVar && !e.hasFVar then
3631             true
3632         else match e with
3633         | Expr.mdata _ b _ => visit b
3634         | Expr.proj _ _ s _ => visit s
3635         | Expr.app f a _ => visit f && visit a
3636         | Expr.lam _ d b _ => visit d && visit b
3637         | Expr.forallE _ d b _ => visit d && visit b
3638         | Expr.letE _ t v b _ => visit t && visit v && visit b
3639         | Expr.bvar .. => true
3640         | Expr.sort .. => true
3641         | Expr.const .. => true
3642         | Expr.lit .. => true
3643         | Expr.fvar fvarId .. =>
3644             if mvarDecl.lctx.contains fvarId then true
3645             else match lctx.find? fvarId with
3646             | some (LocalDecl.ldecl (value := v) ..) => false -- need expensive CheckAssignment.check
3647             | _ =>
3648                 if fvars.any fun x => x.fvarId! == fvarId then true

```

```

3649         else false -- We could throw an exception here, but we would have to use ExceptM. So, we let CheckAssignment.check do it
3650     | Expr.mvar mvarId' _ =>
3651         match mctx.getExprAssignment? mvarId' with
3652         | some _ => false -- use CheckAssignment.check to instantiate
3653         | none _ =>
3654             if mvarId' == mvarId then false -- occurs check failed, use CheckAssignment.check to throw exception
3655             else match mctx.findDecl? mvarId' with
3656             | none _ => false
3657             | some mvarDecl' =>
3658                 if hasCtxLocals then false -- use CheckAssignment.check
3659                 else if mvarDecl'.lctx.isSubPrefixOf mvarDecl'.lctx fvars then true
3660                 else false -- use CheckAssignment.check
3661     visit e
3662
3663 end CheckAssignmentQuick
3664
3665 /--
3666   Auxiliary function for handling constraints of the form `?m a1 ... an == v`.
3667   It will check whether we can perform the assignment
3668   ```
3669   ?m := fun fvars => v
3670   ```
3671   The result is `none` if the assignment can't be performed.
3672   The result is `some newV` where `newV` is a possibly updated `v`. This method may need
3673   to unfold let-declarations. -/
3674 def checkAssignment (mvarId : MVarId) (fvars : Array Expr) (v : Expr) : MetaM (Option Expr) := do
3675   /- Check whether `mvarId` occurs in the type of `fvars` or not. If it does, return `none`
3676   to prevent us from creating the cyclic assignment `?m := fun fvars => v` -/
3677   for fvar in fvars do
3678     unless (← occursCheck mvarId (← inferType fvar)) do
3679       return none
3680   if !v.hasExprMVar && !v.hasFVar then
3681     pure (some v)
3682   else
3683     let mvarDecl ← getMVarDecl mvarId
3684     let hasCtxLocals := fvars.any fun fvar => mvarDecl.lctx.containsFVar fvar
3685     let ctx ← read
3686     let mctx ← getMCtx
3687     if CheckAssignmentQuick.check hasCtxLocals ctx.config.ctxApprox mctx ctx.lctx mvarDecl mvarId fvars v then
3688       pure (some v)
3689     else
3690       let v ← instantiateMVars v
3691       CheckAssignment.checkAssignmentAux mvarId fvars hasCtxLocals v
3692
3693 private def processAssignmentFOApproxAux (mvar : Expr) (args : Array Expr) (v : Expr) : MetaM Bool :=
3694   match v with
3695   | Expr.app f a _ =>

```

```

3696   if args.isEmpty then
3697     pure false
3698   else
3699     Meta.isExprDefEqAux args.back a <&&> Meta.isExprDefEqAux (mkAppRange mvar 0 (args.size - 1) args) f
3700 | _      => pure false
3701
3702 /-
3703 Auxiliary method for applying first-order unification. It is an approximation.
3704 Remark: this method is trying to solve the unification constraint:
3705
3706     ?m a1 ... an =?= v
3707
3708 It is uses processAssignmentFOApproxAux, if it fails, it tries to unfold `v`.
3709
3710 We have added support for unfolding here because we want to be able to solve unification problems such as
3711
3712     ?m Unit =?= ITactic
3713
3714 where `ITactic` is defined as
3715
3716     def ITactic := Tactic Unit
3717 -/
3718 private partial def processAssignmentFOApprox (mvar : Expr) (args : Array Expr) (v : Expr) : MetaM Bool :=
3719   let rec loop (v : Expr) := do
3720     let cfg ← getConfig
3721     if !cfg.foApprox then
3722       pure false
3723     else
3724       trace[Meta.isDefEq.foApprox]! "{mvar} {args} := {v}"
3725       let v := v.headBeta
3726       if (← commitWhen <| processAssignmentFOApproxAux mvar args v) then
3727         pure true
3728       else
3729         match (← unfoldDefinition? v) with
3730         | none    => pure false
3731         | some v  => loop v
3732   loop v
3733
3734 private partial def simpAssignmentArgAux : Expr → MetaM Expr
3735 | Expr.mdata _ e _      => simpAssignmentArgAux e
3736 | e@(Expr.fvar fvarId _) => do
3737   let decl ← getLocalDecl fvarId
3738   match decl.value? with
3739   | some value => simpAssignmentArgAux value
3740   | _         => pure e
3741 | e => pure e
3742

```



```

3743 /- Auxiliary procedure for processing `?m a1 ... an =?= v`.
3744   We apply it to each `ai`. It instantiates assigned metavariables if `ai` is of the form `f[?n] b1 ... bm`,
3745   and then removes metadata, and zeta-expand let-decls. -/
3746 private def simpAssignmentArg (arg : Expr) : MetaM Expr := do
3747   let arg ← if arg.getAppFn.hasExprMVar then instantiateMVars arg else pure arg
3748   simpAssignmentArgAux arg
3749
3750 /- Assign `mvar := fun a1 ... a{numArgs} => v`.
3751   We use it at `processConstApprox` and `isDefEqMVarSelf` -/
3752 private def assignConst (mvar : Expr) (numArgs : Nat) (v : Expr) : MetaM Bool := do
3753   let mvarDecl ← getMVarDecl mvar.mvarId!
3754   forallBoundedTelescope mvarDecl.type numArgs fun xs _ => do
3755     if xs.size != numArgs then
3756       pure false
3757     else
3758       let some v ← mkLambdaFVarsWithLetDeps xs v | pure false
3759       match (← checkAssignment mvar.mvarId! #[] v) with
3760       | none => pure false
3761       | some v =>
3762         trace[Meta.isDefEq.constApprox]! "{mvar} := {v}"
3763         checkTypesAndAssign mvar v
3764
3765 private def processConstApprox (mvar : Expr) (numArgs : Nat) (v : Expr) : MetaM Bool := do
3766   let cfg ← getConfig
3767   let mvarId := mvar.mvarId!
3768   let mvarDecl ← getMVarDecl mvarId
3769   if mvarDecl.numScopeArgs == numArgs || cfg.constApprox then
3770     assignConst mvar numArgs v
3771   else
3772     pure false
3773
3774 /- Tries to solve `?m a1 ... an =?= v` by assigning `?m`.
3775   It assumes `?m` is unassigned. -/
3776 private partial def processAssignment (mvarApp : Expr) (v : Expr) : MetaM Bool :=
3777   traceCtx `Meta.isDefEq.assign` do
3778     trace[Meta.isDefEq.assign]! "{mvarApp} := {v}"
3779     let mvar := mvarApp.getAppFn
3780     let mvarDecl ← getMVarDecl mvar.mvarId!
3781     let rec process (i : Nat) (args : Array Expr) (v : Expr) := do
3782       let cfg ← getConfig
3783       let useFOApprox (args : Array Expr) : MetaM Bool :=
3784         processAssignmentFOApprox mvar args v <||> processConstApprox mvar args.size v
3785       if h : i < args.size then
3786         let arg := args.get (i, h)
3787         let arg ← simpAssignmentArg arg
3788         let args := args.set (i, h) arg
3789         match arg with

```

```

3790 | Expr.fvar fvarId _ =>
3791   if args[0:i].any fun prevArg => prevArg == arg then
3792     useFOApprox args
3793   else if mvarDecl.lctx.contains fvarId && !cfg.quasiPatternApprox then
3794     useFOApprox args
3795   else
3796     process (i+1) args v
3797 | _ =>
3798   useFOApprox args
3799 else
3800   let v ← instantiateMVars v -- enforce A4
3801   if v.getAppFn == mvar then
3802     -- using A6
3803     useFOApprox args
3804   else
3805     let mvarId := mvar.mvarId!
3806     match (← checkAssignment mvarId args v) with
3807     | none => useFOApprox args
3808     | some v => do
3809       trace[Meta.isDefEq.assign.beforeMkLambda]! "{mvar} {args} := {v}"
3810       let some v ← mkLambdaFVarsWithLetDeps args v | return false
3811       if args.any (fun arg => mvarDecl.lctx.containsFVar arg) then
3812         /- We need to type check `v` because abstraction using `mkLambdaFVars` may have produced
3813            a type incorrect term. See discussion at A2 -/
3814         if (← isTypeCorrect v) then
3815           checkTypesAndAssign mvar v
3816         else
3817           trace[Meta.isDefEq.assign.typeError]! "{mvar} := {v}"
3818           useFOApprox args
3819       else
3820         checkTypesAndAssign mvar v
3821   process 0 mvarApp.getAppArgs v
3822
3823 /--
3824   Similar to processAssignment, but if it fails, compute v's whnf and try again.
3825   This helps to solve constraints such as `?m =?= { α := ?m, ... }.α`
3826   Note this is not perfect solution since we still fail occurs check for constraints such as
3827   ```lean
3828   ?m =?= List { α := ?m, β := Nat }.β
3829   ```
3830 -/
3831 private def processAssignment' (mvarApp : Expr) (v : Expr) : MetaM Bool := do
3832   if (← processAssignment mvarApp v) then
3833     return true
3834   else
3835     let vNew ← whnf v
3836     if vNew != v then

```

```

3837     if mvarApp == vNew then
3838         return true
3839     else
3840         processAssignment mvarApp vNew
3841     else
3842         return false
3843
3844 private def isDeltaCandidate? (t : Expr) : MetaM (Option ConstantInfo) := do
3845     match t.getAppFn with
3846     | Expr.const c _ _ =>
3847         match (← getConst? c) with
3848         | r@(some info) => if info.hasValue then return r else return none
3849         | _              => return none
3850     | _ => pure none
3851
3852 /-- Auxiliary method for isDefEqDelta -/
3853 private def isListLevelDefEq (us vs : List Level) : MetaM LBool :=
3854     toLBoolM <| isListLevelDefEqAux us vs
3855
3856 /-- Auxiliary method for isDefEqDelta -/
3857 private def isDefEqLeft (fn : Name) (t s : Expr) : MetaM LBool := do
3858     trace[Meta.isDefEq.delta.unfoldLeft]! fn
3859     toLBoolM <| Meta.isExprDefEqAux t s
3860
3861 /-- Auxiliary method for isDefEqDelta -/
3862 private def isDefEqRight (fn : Name) (t s : Expr) : MetaM LBool := do
3863     trace[Meta.isDefEq.delta.unfoldRight]! fn
3864     toLBoolM <| Meta.isExprDefEqAux t s
3865
3866 /-- Auxiliary method for isDefEqDelta -/
3867 private def isDefEqLeftRight (fn : Name) (t s : Expr) : MetaM LBool := do
3868     trace[Meta.isDefEq.delta.unfoldLeftRight]! fn
3869     toLBoolM <| Meta.isExprDefEqAux t s
3870
3871 /-- Try to solve `f a₁ ... aₙ == f b₁ ... bₙ` by solving `a₁ == b₁, ..., aₙ == bₙ`.
3872
3873     Auxiliary method for isDefEqDelta -/
3874 private def tryHeuristic (t s : Expr) : MetaM Bool :=
3875     let tFn := t.getAppFn
3876     let sFn := s.getAppFn
3877     traceCtx `Meta.isDefEq.delta do
3878         /-
3879         We process arguments before universe levels to reduce a source of brittleness in the TC procedure.
3880
3881         In the TC procedure, we can solve problems containing metavariables.
3882         If the TC procedure tries to assign one of these metavariables, it interrupts the search
3883         using a "stuck" exception. The elaborator catches it, and "interprets" it as "we should try again later".

```

```

3884 Now suppose we have a TC problem, and there are two "local" candidate instances we can try: "bad" and "good".
3885 The "bad" candidate is stuck because of a universe metavariable in the TC problem.
3886 If we try "bad" first, the TC procedure is interrupted. Moreover, if we have ignored the exception,
3887 "bad" would fail anyway trying to assign two different free variables  $\alpha =?= \beta$ .
3888 Example: `Preorder.{?u}  $\alpha =?=$  Preorder.{?v}  $\beta$ , where `?u` and `?v` are universe metavariables that were
3889 not created by the TC procedure.
3890 The key issue here is that we have an `isDefEq t s` invocation that is interrupted by the "stuck" exception,
3891 but it would have failed anyway if we had continued processing it.
3892 By solving the arguments first, we make the example above fail without throwing the "stuck" exception.
3893
3894 TODO: instead of throwing an exception as soon as we get stuck, we should just set a flag.
3895 Then the entry-point for `isDefEq` checks the flag before returning `true`.
3896 -/
3897 commitWhen do
3898   let b ← isDefEqArgs tFn t.getAppArgs s.getAppArgs
3899       <&&>
3900       isListLevelDefEqAux tFn.constLevels! sFn.constLevels!
3901   unless b do
3902     trace[Meta.isDefEq.delta]! "heuristic failed {t} == {s}"
3903   pure b
3904
3905 /-- Auxiliary method for isDefEqDelta -/
3906 private abbrev unfold { $\alpha$ } (e : Expr) (failK : MetaM  $\alpha$ ) (successK : Expr → MetaM  $\alpha$ ) : MetaM  $\alpha$  := do
3907   match (← unfoldDefinition? e) with
3908   | some e => successK e
3909   | none   => failK
3910
3911 /-- Auxiliary method for isDefEqDelta -/
3912 private def unfoldBothDefEq (fn : Name) (t s : Expr) : MetaM LBool := do
3913   match t, s with
3914   | Expr.const _ ls1 _, Expr.const _ ls2 _ => isListLevelDefEq ls1 ls2
3915   | Expr.app _ _ _, Expr.app _ _ _ =>
3916     if (← tryHeuristic t s) then
3917       pure LBool.true
3918     else
3919       unfold t
3920         (unfold s (pure LBool.false) (fun s => isDefEqRight fn t s))
3921         (fun t => unfold s (isDefEqLeft fn t s) (fun s => isDefEqLeftRight fn t s))
3922   | _, _ => pure LBool.false
3923
3924 private def sameHeadSymbol (t s : Expr) : Bool :=
3925   match t.getAppFn, s.getAppFn with
3926   | Expr.const c1 _ _, Expr.const c2 _ _ => true
3927   | _, _ => false
3928
3929 /--
3930 - If headSymbol (unfold t) == headSymbol s, then unfold t

```

```

3931 - If headSymbol (unfold s) == headSymbol t, then unfold s
3932 - Otherwise unfold t and s if possible.
3933
3934 Auxiliary method for isDefEqDelta -/
3935 private def unfoldComparingHeadsDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool :=
3936   unfold t
3937     (unfold s
3938       (pure LBool.undef) -- `t` and `s` failed to be unfolded
3939       (fun s => isDefEqRight sInfo.name t s))
3940   (fun tNew =>
3941     if sameHeadSymbol tNew s then
3942       isDefEqLeft tInfo.name tNew s
3943     else
3944       unfold s
3945         (isDefEqLeft tInfo.name tNew s)
3946         (fun sNew =>
3947           if sameHeadSymbol t sNew then
3948             isDefEqRight sInfo.name t sNew
3949           else
3950             isDefEqLeftRight tInfo.name tNew sNew))
3951
3952 /-- If `t` and `s` do not contain metavariables, then use
3953   kernel definitional equality heuristics.
3954   Otherwise, use `unfoldComparingHeadsDefEq`.
3955
3956   Auxiliary method for isDefEqDelta -/
3957 private def unfoldDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool :=
3958   if !t.hasExprMVar && !s.hasExprMVar then
3959     /- If `t` and `s` do not contain metavariables,
3960       we simulate strategy used in the kernel. -/
3961     if tInfo.hints.lt sInfo.hints then
3962       unfold t (unfoldComparingHeadsDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
3963     else if sInfo.hints.lt tInfo.hints then
3964       unfold s (unfoldComparingHeadsDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
3965     else
3966       unfoldComparingHeadsDefEq tInfo sInfo t s
3967   else
3968     unfoldComparingHeadsDefEq tInfo sInfo t s
3969
3970 /--
3971   When `TransparencyMode` is set to `default` or `all`.
3972   If `t` is reducible and `s` is not ==> `isDefEqLeft` (unfold t) s`
3973   If `s` is reducible and `t` is not ==> `isDefEqRight` t (unfold s)`
3974
3975   Otherwise, use `unfoldDefEq`
3976
3977   Auxiliary method for isDefEqDelta -/

```

```

3978 private def unfoldReducibeDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool := do
3979   if (← shouldReduceReducibleOnly) then
3980     unfoldDefEq tInfo sInfo t s
3981   else
3982     let tReducible ← isReducible tInfo.name
3983     let sReducible ← isReducible sInfo.name
3984     if tReducible && !sReducible then
3985       unfold t (unfoldDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
3986     else if !tReducible && sReducible then
3987       unfold s (unfoldDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
3988     else
3989       unfoldDefEq tInfo sInfo t s
3990
3991 /--
3992   If `t` is a projection function application and `s` is not ==> `isDefEqRight t (unfold s)`
3993   If `s` is a projection function application and `t` is not ==> `isDefEqRight (unfold t) s`
3994
3995   Otherwise, use `unfoldReducibeDefEq`
3996
3997   Auxiliary method for isDefEqDelta -/
3998 private def unfoldNonProjFnDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool := do
3999   let tProj? ← isProjectionFn tInfo.name
4000   let sProj? ← isProjectionFn sInfo.name
4001   if tProj? && !sProj? then
4002     unfold s (unfoldDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
4003   else if !tProj? && sProj? then
4004     unfold t (unfoldDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
4005   else
4006     unfoldReducibeDefEq tInfo sInfo t s
4007
4008 /--
4009   isDefEq by lazy delta reduction.
4010   This method implements many different heuristics:
4011   1- If only `t` can be unfolded => then unfold `t` and continue
4012   2- If only `s` can be unfolded => then unfold `s` and continue
4013   3- If `t` and `s` can be unfolded and they have the same head symbol, then
4014     a) First try to solve unification by unifying arguments.
4015     b) If it fails, unfold both and continue.
4016     Implemented by `unfoldBothDefEq`
4017   4- If `t` is a projection function application and `s` is not => then unfold `s` and continue.
4018   5- If `s` is a projection function application and `t` is not => then unfold `t` and continue.
4019   Remark: 4&5 are implemented by `unfoldNonProjFnDefEq`
4020   6- If `t` is reducible and `s` is not => then unfold `t` and continue.
4021   7- If `s` is reducible and `t` is not => then unfold `s` and continue
4022   Remark: 6&7 are implemented by `unfoldReducibeDefEq`
4023   8- If `t` and `s` do not contain metavariables, then use heuristic used in the Kernel.
4024     Implemented by `unfoldDefEq`

```

```

4025 9- If `headSymbol (unfold t) == headSymbol s`, then unfold t and continue.
4026 10- If `headSymbol (unfold s) == headSymbol t`, then unfold s
4027 11- Otherwise, unfold `t` and `s` and continue.
4028 Remark: 9&10&11 are implemented by `unfoldComparingHeadsDefEq` -/
4029 private def isDefEqDelta (t s : Expr) : MetaM LBool := do
4030   let tInfo? ← isDeltaCandidate? t.getAppFn
4031   let sInfo? ← isDeltaCandidate? s.getAppFn
4032   match tInfo?, sInfo? with
4033   | none,      none      => pure LBool.undef
4034   | some tInfo, none      => unfold t (pure LBool.undef) fun t => isDefEqLeft tInfo.name t s
4035   | none,      some sInfo => unfold s (pure LBool.undef) fun s => isDefEqRight sInfo.name t s
4036   | some tInfo, some sInfo =>
4037     if tInfo.name == sInfo.name then
4038       unfoldBothDefEq tInfo.name t s
4039     else
4040       unfoldNonProjFnDefEq tInfo sInfo t s
4041
4042 private def isAssigned : Expr → MetaM Bool
4043 | Expr.mvar mvarId _ => isExprMVarAssigned mvarId
4044 | _                  => pure false
4045
4046 private def isDelayedAssignedHead (tFn : Expr) (t : Expr) : MetaM Bool := do
4047   match tFn with
4048   | Expr.mvar mvarId _ =>
4049     if (← isDelayedAssigned mvarId) then
4050       let tNew ← instantiateMVars t
4051       return tNew != t
4052     else
4053       pure false
4054 | _ => pure false
4055
4056 private def isSynthetic : Expr → MetaM Bool
4057 | Expr.mvar mvarId _ => do
4058   let mvarDecl ← getMVarDecl mvarId
4059   match mvarDecl.kind with
4060   | MetavarKind.synthetic      => pure true
4061   | MetavarKind.syntheticOpaque => pure true
4062   | MetavarKind.natural        => pure false
4063 | _                            => pure false
4064
4065 private def isAssignable : Expr → MetaM Bool
4066 | Expr.mvar mvarId _ => do let b ← isReadOnlyOrSyntheticOpaqueExprMVar mvarId; pure (!b)
4067 | _                  => pure false
4068
4069 private def etaEq (t s : Expr) : Bool :=
4070   match t.etaExpanded? with
4071   | some t => t == s

```

```

4072 | none => false
4073
4074 private def isLetFVar (fvarId : FVarId) : MetaM Bool := do
4075   let decl ← getLocalDecl fvarId
4076   pure decl.isLet
4077
4078 private def isDefEqProofIrrel (t s : Expr) : MetaM LBool := do
4079   let status ← isProofQuick t
4080   match status with
4081   | LBool.false =>
4082     pure LBool.undef
4083   | LBool.true =>
4084     let tType ← inferType t
4085     let sType ← inferType s
4086     toLBoolM <| Meta.isExprDefEqAux tType sType
4087   | LBool.undef =>
4088     let tType ← inferType t
4089     if (← isProp tType) then
4090       let sType ← inferType s
4091       toLBoolM <| Meta.isExprDefEqAux tType sType
4092     else
4093       pure LBool.undef
4094
4095 /- Try to solve constraint of the form `?m args₁ =?= ?m args₂`.
4096 - First try to unify `args₁` and `args₂`, and return true if successful
4097 - Otherwise, try to assign `?m` to a constant function of the form `fun x_1 ... x_n => ?n`
4098   where `?n` is a fresh metavariable. See `processConstApprox`. -/
4099 private def isDefEqMVarSelf (mvar : Expr) (args₁ args₂ : Array Expr) : MetaM Bool := do
4100   if args₁.size != args₂.size then
4101     pure false
4102   else if (← isDefEqArgs mvar args₁ args₂) then
4103     pure true
4104   else if !(← isAssignable mvar) then
4105     pure false
4106   else
4107     let cfg ← getConfig
4108     let mvarId := mvar.mvarId!
4109     let mvarDecl ← getMVarDecl mvarId
4110     if mvarDecl.numScopeArgs == args₁.size || cfg.constApprox then
4111       let type ← inferType (mkAppN mvar args₁)
4112       let auxMVar ← mkAuxMVar mvarDecl.lctx mvarDecl.localInstances type
4113       assignConst mvar args₁.size auxMVar
4114     else
4115       pure false
4116
4117 /- Remove unnecessary let-decls -/
4118 private def consumeLet : Expr → Expr

```



```

4119 | e@(Expr.letE _ _ _ b _) => if b.hasLooseBVars then e else consumeLet b
4120 | e                        => e
4121
4122 mutual
4123
4124 private partial def isDefEqQuick (t s : Expr) : MetaM LBool :=
4125   let t := consumeLet t
4126   let s := consumeLet s
4127   match t, s with
4128   | Expr.lit l1 _,      Expr.lit l2 _      => return (l1 == l2).toLBool
4129   | Expr.sort u _,      Expr.sort v _      => toLBoolM <| isLevelDefEqAux u v
4130   | Expr.lam ..,        Expr.lam ..        => if t == s then pure LBool.true else toLBoolM <| isDefEqBinding t s
4131   | Expr.forallE ..,    Expr.forallE ..    => if t == s then pure LBool.true else toLBoolM <| isDefEqBinding t s
4132   | Expr.mdata _ t _,   s                  => isDefEqQuick t s
4133   | t,                  Expr.mdata _ s _    => isDefEqQuick t s
4134   | Expr.fvar fvarId1 _, Expr.fvar fvarId2 _ => do
4135     if (← isLetFVar fvarId1 <||> isLetFVar fvarId2) then
4136       pure LBool.undef
4137     else if fvarId1 == fvarId2 then
4138       pure LBool.true
4139     else
4140       isDefEqProofIrrel t s
4141   | t, s =>
4142     isDefEqQuickOther t s
4143
4144 private partial def isDefEqQuickOther (t s : Expr) : MetaM LBool := do
4145   if t == s then
4146     pure LBool.true
4147   else if etaEq t s || etaEq s t then
4148     pure LBool.true -- t ==?= (fun xs => t xs)
4149   else
4150     let tFn := t.getAppFn
4151     let sFn := s.getAppFn
4152     if !tFn.isMVar && !sFn.isMVar then
4153       pure LBool.undef
4154     else if (← isAssigned tFn) then
4155       let t ← instantiateMVars t
4156       isDefEqQuick t s
4157     else if (← isAssigned sFn) then
4158       let s ← instantiateMVars s
4159       isDefEqQuick t s
4160     else if (← isDelayedAssignedHead tFn t) then
4161       let t ← instantiateMVars t
4162       isDefEqQuick t s
4163     else if (← isDelayedAssignedHead sFn s) then
4164       let s ← instantiateMVars s
4165       isDefEqQuick t s

```

```

4166   else if (← isSynthetic tFn <&& trySynthPending tFn) then
4167     let t ← instantiateMVars t
4168     isDefEqQuick t s
4169   else if (← isSynthetic sFn <&& trySynthPending sFn) then
4170     let s ← instantiateMVars s
4171     isDefEqQuick t s
4172   else if tFn.isMVar && sFn.isMVar && tFn == sFn then
4173     Bool.toLBool <$> isDefEqMVarSelf tFn t.getAppArgs s.getAppArgs
4174   else
4175     let tAssign? ← isAssignable tFn
4176     let sAssign? ← isAssignable sFn
4177     let assignableMsg (b : Bool) := if b then "[assignable]" else "[nonassignable]"
4178     trace[Meta.isDefEq]! "{t} {assignableMsg tAssign?} =?= {s} {assignableMsg sAssign?}"
4179     if tAssign? && !sAssign? then
4180       toLBoolM <| processAssignment' t s
4181     else if !tAssign? && sAssign? then
4182       toLBoolM <| processAssignment' s t
4183     else if !tAssign? && !sAssign? then
4184       if tFn.isMVar || sFn.isMVar then
4185         let ctx ← read
4186         if ctx.config.isDefEqStuckEx then do
4187           trace[Meta.isDefEq.stuck]! "{t} =?= {s}"
4188           Meta.throwIsDefEqStuck
4189       else
4190         pure LBool.false
4191     else
4192       pure LBool.undef
4193   else
4194     isDefEqQuickMVarMVar t s
4195
4196 -- Both `t` and `s` are terms of the form `?m ...`
4197 private partial def isDefEqQuickMVarMVar (t s : Expr) : MetaM LBool := do
4198   let tFn := t.getAppFn
4199   let sFn := s.getAppFn
4200   let tMVarDecl ← getMVarDecl tFn.mvarId!
4201   let sMVarDecl ← getMVarDecl sFn.mvarId!
4202   if s.isMVar && !t.isMVar then
4203     /- Solve `?m t =?= ?n` by trying first `?n := ?m t`.
4204     Reason: this assignment is precise. -/
4205     if (← commitWhen (processAssignment s t)) then
4206       pure LBool.true
4207     else
4208       toLBoolM <| processAssignment t s
4209   else
4210     if (← commitWhen (processAssignment t s)) then
4211       pure LBool.true
4212     else

```

```

4213         toLBoolM <| processAssignment s t
4214
4215 end
4216
4217 @[inline] def whenUndefDo (x : MetaM LBool) (k : MetaM Bool) : MetaM Bool := do
4218   let status ← x
4219   match status with
4220   | LBool.true  => pure true
4221   | LBool.false => pure false
4222   | LBool.undef => k
4223
4224 @[specialize] private def unstuckMVar (e : Expr) (successK : Expr → MetaM Bool) (failK : MetaM Bool): MetaM Bool := do
4225   match (← getStuckMVar? e) with
4226   | some mvarId =>
4227     trace[Meta.isDefEq.stuckMVar]! "found stuck MVar {mkMVar mvarId} : {← inferType (mkMVar mvarId)}"
4228     if (← Meta.synthPending mvarId) then
4229       let e ← instantiateMVars e
4230       successK e
4231     else
4232       failK
4233   | none => failK
4234
4235 private def isDefEqOnFailure (t s : Expr) : MetaM Bool :=
4236   unstuckMVar t (fun t => Meta.isExprDefEqAux t s) <|
4237   unstuckMVar s (fun s => Meta.isExprDefEqAux t s) <|
4238   tryUnificationHints t s <||> tryUnificationHints s t
4239
4240 private def isDefEqProj : Expr → Expr → MetaM Bool
4241 | Expr.proj _ i t _, Expr.proj _ j s _ => pure (i == j) <&&> Meta.isExprDefEqAux t s
4242 | _, _ => pure false
4243
4244 /-
4245   Given applications `t` and `s` that are in WHNF (modulo the current transparency setting),
4246   check whether they are definitionally equal or not.
4247 -/
4248 private def isDefEqApp (t s : Expr) : MetaM Bool := do
4249   let tFn := t.getAppFn
4250   let sFn := s.getAppFn
4251   if tFn.isConst && sFn.isConst && tFn.constName! == sFn.constName! then
4252     /- See comment at `tryHeuristic` explaining why we processe arguments before universe levels. -/
4253     if (← commitWhen (isDefEqArgs tFn t.getAppArgs s.getAppArgs <&&> isListLevelDefEqAux tFn.constLevels! sFn.constLevels!)) then
4254       return true
4255     else
4256       isDefEqOnFailure t s
4257   else if (← commitWhen (Meta.isExprDefEqAux tFn sFn s.getAppFn <&&> isDefEqArgs tFn t.getAppArgs s.getAppArgs)) then
4258     return true
4259   else

```

```

4260     isDefEqOnFailure t s
4261
4262 partial def isExprDefEqAuxImpl (t : Expr) (s : Expr) : MetaM Bool := do
4263   trace[Meta.isDefEq.step]! "{t} == {s}"
4264   checkMaxHeartbeats "isDefEq"
4265   withNestedTraces do
4266     whenUndefDo (isDefEqQuick t s) do
4267       whenUndefDo (isDefEqProofIrrel t s) do
4268         let t' ← whnfCore t
4269         let s' ← whnfCore s
4270         if t != t' || s != s' then
4271           isExprDefEqAuxImpl t' s'
4272         else do
4273           if (← (isDefEqEta t s <||> isDefEqEta s t)) then pure true else
4274             if (← isDefEqProj t s) then pure true else
4275               whenUndefDo (isDefEqNative t s) do
4276                 whenUndefDo (isDefEqNat t s) do
4277                   whenUndefDo (isDefEqOffset t s) do
4278                     whenUndefDo (isDefEqDelta t s) do
4279                       if t.isConst && s.isConst then
4280                         if t.constName! == s.constName! then isListLevelDefEqAux t.constLevels! s.constLevels! else pure false
4281                       else if t.isApp && s.isApp then
4282                         isDefEqApp t s
4283                       else
4284                         whenUndefDo (isDefEqStringLit t s) do
4285                           isDefEqOnFailure t s
4286
4287 builtin_initialize
4288   isExprDefEqAuxRef.set isExprDefEqAuxImpl
4289
4290 builtin_initialize
4291   registerTraceClass `Meta.isDefEq
4292   registerTraceClass `Meta.isDefEq.foApprox
4293   registerTraceClass `Meta.isDefEq.constApprox
4294   registerTraceClass `Meta.isDefEq.delta
4295   registerTraceClass `Meta.isDefEq.step
4296   registerTraceClass `Meta.isDefEq.assign
4297
4298 end Lean.Meta
4299 // ::::::::::::::
4300 // ForEachExpr.lean
4301 // ::::::::::::::
4302 /-
4303 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
4304 Released under Apache 2.0 license as described in the file LICENSE.
4305 Authors: Leonardo de Moura
4306 -/

```

```

4307 import Lean.Expr
4308 import Lean.Util.MonadCache
4309 import Lean.Meta.Basic
4310
4311 namespace Lean.Meta
4312 namespace ForEachExpr
4313
4314 abbrev M := MonadCacheT Expr Unit MetaM
4315
4316 mutual
4317
4318 private partial def visitBinder (fn : Expr → MetaM Bool) : Array Expr → Nat → Expr → M Unit
4319 | fvars, j, Expr.lam n d b c => do
4320   let d := d.instantiateRevRange j fvars.size fvars;
4321   visit fn d;
4322   withLocalDecl n c.binderInfo d fun x =>
4323     visitBinder fn (fvars.push x) j b
4324 | fvars, j, Expr.forallE n d b c => do
4325   let d := d.instantiateRevRange j fvars.size fvars;
4326   visit fn d;
4327   withLocalDecl n c.binderInfo d fun x =>
4328     visitBinder fn (fvars.push x) j b
4329 | fvars, j, Expr.letE n t v b _ => do
4330   let t := t.instantiateRevRange j fvars.size fvars;
4331   visit fn t;
4332   let v := v.instantiateRevRange j fvars.size fvars;
4333   visit fn v;
4334   withLetDecl n t v fun x =>
4335     visitBinder fn (fvars.push x) j b
4336 | fvars, j, e => visit fn $ e.instantiateRevRange j fvars.size fvars
4337
4338 partial def visit (fn : Expr → MetaM Bool) (e : Expr) : M Unit :=
4339   checkCache e fun _ => do
4340     if (← liftM (fn e)) then
4341       match e with
4342       | Expr.forallE _ _ _ _ => visitBinder fn #[] 0 e
4343       | Expr.lam _ _ _ _ => visitBinder fn #[] 0 e
4344       | Expr.letE _ _ _ _ => visitBinder fn #[] 0 e
4345       | Expr.app f a _ => visit fn f; visit fn a
4346       | Expr.mdata _ b _ => visit fn b
4347       | Expr.proj _ _ b _ => visit fn b
4348       | _ => pure ()
4349
4350 end
4351
4352 end ForEachExpr
4353

```

```

4354 /-- Similar to `Expr.forEach`, but creates free variables whenever going inside of a binder. -/
4355 def forEachExpr' (e : Expr) (f : Expr → MetaM Bool) : MetaM Unit :=
4356   ForEachExpr.visit f e |>.run
4357
4358 /-- Similar to `Expr.forEach`, but creates free variables whenever going inside of a binder. -/
4359 def forEachExpr (e : Expr) (f : Expr → MetaM Unit) : MetaM Unit :=
4360   forEachExpr' e fun e => do
4361     f e
4362     pure true
4363
4364 end Lean.Meta
4365 // ::::::::::::::
4366 // FunInfo.lean
4367 // ::::::::::::::
4368 /-
4369 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
4370 Released under Apache 2.0 license as described in the file LICENSE.
4371 Authors: Leonardo de Moura
4372 -/
4373 import Lean.Meta.Basic
4374 import Lean.Meta.InferType
4375
4376 namespace Lean.Meta
4377
4378 @[inline] private def checkFunInfoCache (fn : Expr) (maxArgs? : Option Nat) (k : MetaM FunInfo) : MetaM FunInfo := do
4379   let s ← get
4380   let t ← getTransparency
4381   match s.cache.funInfo.find? (t, fn, maxArgs?) with
4382   | some finfo => pure finfo
4383   | none      => do
4384     let finfo ← k
4385     modify fun s => { s with cache := { s.cache with funInfo := s.cache.funInfo.insert (t, fn, maxArgs?) finfo } }
4386     pure finfo
4387
4388 @[inline] private def whenHasVar {α} (e : Expr) (deps : α) (k : α → α) : α :=
4389   if e.hasFVar then k deps else deps
4390
4391 private def collectDeps (fvars : Array Expr) (e : Expr) : Array Nat :=
4392   let rec visit : Expr → Array Nat → Array Nat
4393     | e@(Expr.app f a _),      deps => whenHasVar e deps (visit a ◦ visit f)
4394     | e@(Expr.forallE _ d b _), deps => whenHasVar e deps (visit b ◦ visit d)
4395     | e@(Expr.lam _ d b _),    deps => whenHasVar e deps (visit b ◦ visit d)
4396     | e@(Expr.letE _ t v b _), deps => whenHasVar e deps (visit b ◦ visit v ◦ visit t)
4397     | Expr.proj _ _ e _,      deps => visit e deps
4398     | Expr.mdata _ e _,       deps => visit e deps
4399     | e@(Expr.fvar _ _),      deps =>
4400       match fvars.indexOf? e with

```

```

4401     | none    => deps
4402     | some i => if deps.contains i.val then deps else deps.push i.val
4403     | _,      => deps => deps
4404   let deps := visit e #[]
4405   deps.qsort (fun i j => i < j)
4406
4407 /-- Update `hasFwdDeps` fields using new `backDeps` -/
4408 private def updateHasFwdDeps (pinfo : Array ParamInfo) (backDeps : Array Nat) : Array ParamInfo :=
4409   if backDeps.size == 0 then
4410     pinfo
4411   else
4412     -- update hasFwdDeps fields
4413     pinfo.mapIdx fun i info =>
4414       if info.hasFwdDeps then info
4415       else if backDeps.contains i then
4416         { info with hasFwdDeps := true }
4417       else
4418         info
4419
4420 private def getFunInfoAux (fn : Expr) (maxArgs? : Option Nat) : MetaM FunInfo :=
4421   checkFunInfoCache fn maxArgs? do
4422     let fnType ← inferType fn
4423     withTransparency TransparencyMode.default do
4424       forallBoundedTelescope fnType maxArgs? fun fvars type => do
4425         let mut pinfo := #[]
4426         for i in [:fvars.size] do
4427           let fvar := fvars[i]
4428           let decl ← getFVarLocalDecl fvar
4429           let backDeps := collectDeps fvars decl.type
4430           pinfo := updateHasFwdDeps pinfo backDeps
4431           pinfo := pinfo.push {
4432             backDeps      := backDeps,
4433             implicit      := decl.binderInfo == BinderInfo.implicit,
4434             instImplicit   := decl.binderInfo == BinderInfo.instImplicit
4435           }
4436         let resultDeps := collectDeps fvars type
4437         let pinfo      := updateHasFwdDeps pinfo resultDeps
4438         pure { resultDeps := resultDeps, paramInfo := pinfo }
4439
4440 def getFunInfo (fn : Expr) : MetaM FunInfo :=
4441   getFunInfoAux fn none
4442
4443 def getFunInfoNArgs (fn : Expr) (nargs : Nat) : MetaM FunInfo :=
4444   getFunInfoAux fn (some nargs)
4445
4446 end Lean.Meta
4447 // ::::::::::::::

```

```

4448 // GeneralizeTelescope.lean
4449 // ::::::::::::::
4450 /-
4451 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
4452 Released under Apache 2.0 license as described in the file LICENSE.
4453 Authors: Leonardo de Moura
4454 -/
4455 import Lean.Meta.KAbstract
4456
4457 namespace Lean.Meta
4458 namespace GeneralizeTelescope
4459
4460 structure Entry where
4461   expr      : Expr
4462   type      : Expr
4463   modified  : Bool
4464
4465 partial def updateTypes (e eNew : Expr) (entries : Array Entry) (i : Nat) : MetaM (Array Entry) :=
4466   if h : i < entries.size then
4467     let entry := entries.get {i, h}
4468     match entry with
4469     | (_, type, _) => do
4470       let typeAbst ← kabstract type e
4471       if typeAbst.hasLooseBVars then do
4472         let typeNew := typeAbst.instantiate1 eNew
4473         let entries := entries.set {i, h} { entry with type := typeNew, modified := true }
4474         updateTypes e eNew entries (i+1)
4475       else
4476         updateTypes e eNew entries (i+1)
4477   else
4478     pure entries
4479
4480 partial def generalizeTelescopeAux {α} (k : Array Expr → MetaM α)
4481   (entries : Array Entry) (i : Nat) (fvars : Array Expr) : MetaM α := do
4482   if h : i < entries.size then
4483     let replace (baseUserName : Name) (e : Expr) (type : Expr) : MetaM α := do
4484       let userName ← mkFreshUserName baseUserName
4485       withLocalDeclD userName type fun x => do
4486         let entries ← updateTypes e x entries (i+1)
4487         generalizeTelescopeAux k entries (i+1) (fvars.push x)
4488     match entries.get {i, h} with
4489     | (e@(Expr.fvar fvarId _), type, false) =>
4490       let localDecl ← getLocalDecl fvarId
4491       match localDecl with
4492       | LocalDecl.cdecl .. => generalizeTelescopeAux k entries (i+1) (fvars.push e)
4493       | LocalDecl.ldecl .. => replace localDecl.userName e type
4494     | (e, type, modified) =>

```



```

4495     if modified then
4496         unless (← isTypeCorrect type) do
4497             throwError! "failed to create telescope generalizing {entries.map Entry.expr}"
4498     replace `x e type
4499 else
4500     k fvars
4501
4502 end GeneralizeTelescope
4503
4504 open GeneralizeTelescope
4505
4506 /--
4507   Given expressions `es := #[e_1, e_2, ..., e_n]`, execute `k` with the
4508   free variables `(x_1 : A_1) (x_2 : A_2 [x_1]) ... (x_n : A_n [x_1, ... x_{n-1}])`.
4509   Moreover,
4510   - type of `e_1` is definitionally equal to `A_1`,
4511   - type of `e_2` is definitionally equal to `A_2[e_1]`.
4512   - ...
4513   - type of `e_n` is definitionally equal to `A_n[e_1, ..., e_{n-1}]`.
4514
4515   This method tries to avoid the creation of new free variables. For example, if `e_i` is a
4516   free variable `x_i` and it is not a let-declaration variable, and its type does not depend on
4517   previous `e_j`'s, the method will just use `x_i`.
4518
4519   The telescope `x_1 ... x_n` can be used to create lambda and forall abstractions.
4520   Moreover, for any type correct lambda abstraction `f` constructed using `mkForall #[x_1, ..., x_n] ...`,
4521   The application `f e_1 ... e_n` is also type correct.
4522
4523   The `kabtract` method is used to "locate" and abstract forward dependencies.
4524   That is, an occurrence of `e_i` in the of `e_j` for `j > i`.
4525
4526   The method checks whether the abstract types `A_i` are type correct. Here is an example
4527   where `generalizeTelescope` fails to create the telescope `x_1 ... x_n`.
4528   Assume the local context contains `(n : Nat := 10) (xs : Vec Nat n) (ys : Vec Nat 10) (h : xs = ys)`.
4529   Then, assume we invoke `generalizeTelescope` with `es := #[10, xs, ys, h]`
4530   A type error is detected when processing `h`'s type. At this point, the method had successfully produced
4531   ```
4532   (x_1 : Nat) (xs : Vec Nat n) (x_2 : Vec Nat x_1)
4533   ```
4534   and the type for the new variable abstracting `h` is `xs = x_2` which is not type correct. -/
4535 def generalizeTelescope {α} (es : Array Expr) (k : Array Expr → MetaM α) : MetaM α := do
4536   let es ← es.mapM fun e => do
4537     let type ← inferType e
4538     let type ← instantiateMVars type
4539     pure { expr := e, type := type, modified := false : Entry }
4540   generalizeTelescopeAux k es 0 #[]
4541

```

```

4542 end Lean.Meta
4543 // ::::::::::::::
4544 // GetConst.lean
4545 // ::::::::::::::
4546 /-
4547 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
4548 Released under Apache 2.0 license as described in the file LICENSE.
4549 Authors: Leonardo de Moura
4550 -/
4551 import Lean.Meta.Instances
4552
4553 namespace Lean.Meta
4554
4555 private def getDefInfo (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
4556   match (← read).config.transparency with
4557   | TransparencyMode.all => return some info
4558   | TransparencyMode.default => return some info
4559   | m =>
4560     if (← isReducible info.name) then
4561       return some info
4562     else if m == TransparencyMode.instances && isGlobalInstance (← getEnv) info.name then
4563       return some info
4564     else
4565       return none
4566
4567 def getConst? (constName : Name) : MetaM (Option ConstantInfo) := do
4568   let env ← getEnv
4569   match env.find? constName with
4570   | some (info@(ConstantInfo.thmInfo _)) => getTheoremInfo info
4571   | some (info@(ConstantInfo.defnInfo _)) => getDefInfo info
4572   | some info => pure (some info)
4573   | none => throwUnknownConstant constName
4574
4575 def getConstNoEx? (constName : Name) : MetaM (Option ConstantInfo) := do
4576   let env ← getEnv
4577   match env.find? constName with
4578   | some (info@(ConstantInfo.thmInfo _)) => getTheoremInfo info
4579   | some (info@(ConstantInfo.defnInfo _)) => getDefInfo info
4580   | some info => pure (some info)
4581   | none => pure none
4582
4583 end Meta
4584 // ::::::::::::::
4585 // Inductive.lean
4586 // ::::::::::::::
4587 /-
4588 Copyright (c) 2020 Microsoft Corporation. All rights reserved.

```

```

4589 Released under Apache 2.0 license as described in the file LICENSE.
4590 Authors: Leonardo de Moura
4591 -/
4592 import Lean.Meta.ExprDefEq
4593
4594 /- Helper methods for inductive datatypes -/
4595
4596 namespace Lean.Meta
4597
4598 /- Return true if the types of the given constructors are compatible. -/
4599 def compatibleCtors (ctorName1 ctorName2 : Name) : MetaM Bool := do
4600   let ctorInfo1 ← getConstInfoCtor ctorName1
4601   let ctorInfo2 ← getConstInfoCtor ctorName2
4602   if ctorInfo1.induct != ctorInfo2.induct then
4603     return false
4604   else
4605     let (_, _, ctorType1) ← forallMetaTelescope ctorInfo1.type
4606     let (_, _, ctorType2) ← forallMetaTelescope ctorInfo2.type
4607     isDefEq ctorType1 ctorType2
4608
4609 end Lean.Meta
4610 // ::::::::::::::
4611 // InferType.lean
4612 // ::::::::::::::
4613 /-
4614 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
4615 Released under Apache 2.0 license as described in the file LICENSE.
4616 Authors: Leonardo de Moura
4617 -/
4618 import Lean.Data.LBool
4619 import Lean.Meta.Basic
4620
4621 namespace Lean
4622
4623 /-
4624 Auxiliary function for instantiating the loose bound variables in `e` with `args[start:stop]`.
4625 This function is similar to `instantiateRevRange`, but it applies beta-reduction when
4626 we instantiate a bound variable with a lambda expression.
4627 Example: Given the term `#0 a`, and `start := 0, stop := 1, args := #[fun x => x]` the result is
4628 `a` instead of `(fun x => x) a`.
4629 This reduction is useful when we are inferring the type of eliminator-like applications.
4630 For example, given `(n m : Nat) (f : Nat → Nat) (h : m = n)`,
4631 the type of `Eq.subst (motive := fun x => f m = f x) h rfl`
4632 is `motive n` which is `(fun (x : Nat) => f m = f x) n`
4633 This function reduces the new application to `f m = f n`
4634
4635 We use it to implement `inferAppType`

```

```

4636 -/
4637 partial def Expr.instantiateBetaRevRange (e : Expr) (start : Nat) (stop : Nat) (args : Array Expr) : Expr :=
4638   if e.hasLooseBVars && stop > start then
4639     assert! stop ≤ args.size
4640     visit e 0 |>.run
4641   else
4642     e
4643 where
4644   visit (e : Expr) (offset : Nat) : MonadStateCacheT (Expr × Nat) Expr Id Expr :=
4645     if offset >= e.looseBVarRange then
4646       -- `e` doesn't have free variables
4647       return e
4648     else checkCache (e, offset) fun _ => do
4649       match e with
4650       | Expr.forallE _ d b _ => return e.updateForallE! (← visit d offset) (← visit b (offset+1))
4651       | Expr.lam _ d b _ => return e.updateLambdaE! (← visit d offset) (← visit b (offset+1))
4652       | Expr.letE _ t v b _ => return e.updateLet! (← visit t offset) (← visit v offset) (← visit b (offset+1))
4653       | Expr.mdata _ b _ => return e.updateMData! (← visit b offset)
4654       | Expr.proj _ _ b _ => return e.updateProj! (← visit b offset)
4655       | Expr.app f a _ =>
4656         e.withAppRev fun f revArgs => do
4657           let fNew ← visit f offset
4658           let revArgs ← revArgs.mapM (visit · offset)
4659           if f.isBVar then
4660             -- try to beta reduce if `f` was a bound variable
4661             return fNew.betaRev revArgs
4662           else
4663             return mkAppRev fNew revArgs
4664       | Expr.bvar vidx _ =>
4665         -- Recall that looseBVarRange for `Expr.bvar` is `vidx+1`.
4666         -- So, we must have offset ≤ vidx, since we are in the "else" branch of `if offset >= e.looseBVarRange`
4667         let n := stop - start
4668         if vidx < offset + n then
4669           return args[stop - (vidx - offset) - 1].liftLooseBVars 0 offset
4670         else
4671           return mkBVar (vidx - n)
4672       -- The following cases are unreachable because they never contain loose bound variables
4673       | Expr.const .. => unreachable!
4674       | Expr.fvar .. => unreachable!
4675       | Expr.mvar .. => unreachable!
4676       | Expr.sort .. => unreachable!
4677       | Expr.lit .. => unreachable!
4678
4679 namespace Meta
4680
4681 def throwFunctionExpected {α} (f : Expr) : MetaM α :=
4682   throwError! "function expected{indentExpr f}"

```

```

4683
4684 private def inferAppType (f : Expr) (args : Array Expr) : MetaM Expr := do
4685   let mut fType ← inferType f
4686   let mut j := 0
4687   /- TODO: check whether `instantiateBetaRevRange` is too expensive, and
4688     use it only when `args` contains a lambda expression. -/
4689   for i in [:args.size] do
4690     match fType with
4691     | Expr.forallE _ _ b _ => fType := b
4692     | _ =>
4693       match (← whnf <| fType.instantiateBetaRevRange j i args) with
4694       | Expr.forallE _ _ b _ => j := i; fType := b
4695       | _ => throwFunctionExpected <| mkAppRange f 0 (i+1) args
4696   return fType.instantiateBetaRevRange j args.size args
4697
4698 def throwIncorrectNumberOfLevels {α} (constName : Name) (us : List Level) : MetaM α :=
4699   throwError! "incorrect number of universe levels {mkConst constName us}"
4700
4701 private def inferConstType (c : Name) (us : List Level) : MetaM Expr := do
4702   let cinfo ← getConstInfo c
4703   if cinfo.levelParams.length == us.length then
4704     return cinfo.instantiateTypeLevelParams us
4705   else
4706     throwIncorrectNumberOfLevels c us
4707
4708 private def inferProjType (structName : Name) (idx : Nat) (e : Expr) : MetaM Expr := do
4709   let failed {α} : Unit → MetaM α := fun _ =>
4710     throwError! "invalid projection{indentExpr (mkProj structName idx e)}"
4711   let structType ← inferType e
4712   let structType ← whnf structType
4713   matchConstStruct structType.getAppFn failed fun structVal structLvls ctorVal =>
4714     let n := structVal.numParams
4715     let structParams := structType.getAppArgs
4716     if n != structParams.size then failed ()
4717     else do
4718       let mut ctorType ← inferAppType (mkConst ctorVal.name structLvls) structParams
4719       for i in [:idx] do
4720         ctorType ← whnf ctorType
4721         match ctorType with
4722         | Expr.forallE _ _ body _ =>
4723           if body.hasLooseBVars then
4724             ctorType := body.instantiate1 $ mkProj structName i e
4725           else
4726             ctorType := body
4727         | _ => failed ()
4728   ctorType ← whnf ctorType
4729   match ctorType with

```

```

4730      | Expr.forallE _ d _ => pure d
4731      | _                  => failed ()
4732
4733 def throwTypeExcepted {α} (type : Expr) : MetaM α :=
4734   throwError! "type expected{indentExpr type}"
4735
4736 def getLevel (type : Expr) : MetaM Level := do
4737   let typeType ← inferType type
4738   let typeType ← whnfD typeType
4739   match typeType with
4740   | Expr.sort lvl _ => pure lvl
4741   | Expr.mvar mvarId _ =>
4742     if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
4743       throwTypeExcepted type
4744     else
4745       let lvl ← mkFreshLevelMVar
4746       assignExprMVar mvarId (mkSort lvl)
4747       pure lvl
4748   | _ => throwTypeExcepted type
4749
4750 private def inferForallType (e : Expr) : MetaM Expr :=
4751   forallTelescope e fun xs e => do
4752     let lvl ← getLevel e
4753     let lvl ← xs.foldrM (init := lvl) fun x lvl => do
4754       let xType ← inferType x
4755       let xTypeLvl ← getLevel xType
4756       pure $ mkLevelIMax' xTypeLvl lvl
4757     pure $ mkSort lvl.normalize
4758
4759 /- Infer type of lambda and let expressions -/
4760 private def inferLambdaType (e : Expr) : MetaM Expr :=
4761   lambdaLetTelescope e fun xs e => do
4762     let type ← inferType e
4763     mkForallFVars xs type
4764
4765 @[inline] private def withLocalDecl' {α} (name : Name) (bi : BinderInfo) (type : Expr) (x : Expr → MetaM α) : MetaM α :=
4766   savingCache do
4767     let fvarId ← mkFreshId
4768     withReader (fun ctx => { ctx with lctx := ctx.lctx.mkLocalDecl fvarId name type bi }) do
4769       x (mkFVar fvarId)
4770
4771 def throwUnknownMVar {α} (mvarId : MVarId) : MetaM α :=
4772   throwError! "unknown metavariable '{mkMVar mvarId}'"
4773
4774 private def inferMVarType (mvarId : MVarId) : MetaM Expr := do
4775   match (← getMCtx).findDecl? mvarId with
4776   | some d => pure d.type

```

```

4777 | none => throwUnknownMVar mvarId
4778
4779 private def inferFVarType (fvarId : FVarId) : MetaM Expr := do
4780   match (← getLCtx).find? fvarId with
4781   | some d => pure d.type
4782   | none   => throwUnknownFVar fvarId
4783
4784 @[inline] private def checkInferTypeCache (e : Expr) (inferType : MetaM Expr) : MetaM Expr := do
4785   match (← get).cache.inferType.find? e with
4786   | some type => pure type
4787   | none      =>
4788     let type ← inferType
4789     modifyInferTypeCache fun c => c.insert e type
4790     pure type
4791
4792 def inferTypeImp (e : Expr) : MetaM Expr :=
4793   let rec infer : Expr → MetaM Expr
4794     | Expr.const c [] _ => inferConstType c []
4795     | Expr.const c us _ => checkInferTypeCache e (inferConstType c us)
4796     | e@(Expr.proj n i s _) => checkInferTypeCache e (inferProjType n i s)
4797     | e@(Expr.app f _ _) => checkInferTypeCache e (inferAppType f.getAppFn e.getAppArgs)
4798     | Expr.mvar mvarId _ => inferMVarType mvarId
4799     | Expr.fvar fvarId _ => inferFVarType fvarId
4800     | Expr.bvar bidx _ => throwError! "unexpected bound variable {mkBVar bidx}"
4801     | Expr.mdata _ e _ => infer e
4802     | Expr.lit v _ => pure v.type
4803     | Expr.sort lvl _ => pure $ mkSort (mkLevelSucc lvl)
4804     | e@(Expr.forallE _ _ _ _) => checkInferTypeCache e (inferForallType e)
4805     | e@(Expr.lam _ _ _ _) => checkInferTypeCache e (inferLambdaType e)
4806     | e@(Expr.letE _ _ _ _) => checkInferTypeCache e (inferLambdaType e)
4807   withTransparency TransparencyMode.default (infer e)
4808
4809 @[builtinInit] def setInferTypeRef : IO Unit :=
4810   inferTypeRef.set inferTypeImp
4811
4812 /--
4813   Return `LBool.true` if given level is always equivalent to universe level zero.
4814   It is used to implement `isProp`. -/
4815 private def isAlwaysZero : Level → Bool
4816   | Level.zero _ => true
4817   | Level.mvar _ _ => false
4818   | Level.param _ _ => false
4819   | Level.succ _ _ => false
4820   | Level.max u v _ => isAlwaysZero u && isAlwaysZero v
4821   | Level.imax _ u _ => isAlwaysZero u
4822
4823 /--

```

```

4824 `isArrowProp type n` is an "approximate" predicate which returns `LBool.true`
4825   if `type` is of the form `A_1 -> ... -> A_n -> Prop`.
4826   Remark: `type` can be a dependent arrow. -/
4827 private partial def isArrowProp : Expr → Nat → MetaM LBool
4828 | Expr.sort u _,      0    => return isAlwaysZero (← instantiateLevelMVars u) |>.toLBool
4829 | Expr.forallE _ _ _ _, 0    => pure LBool.false
4830 | Expr.forallE _ _ b _, n+1 => isArrowProp b n
4831 | Expr.letE _ _ _ b _,   n   => isArrowProp b n
4832 | Expr.mdata _ e _,      n   => isArrowProp e n
4833 | _,                    _    => pure LBool.undef
4834
4835 /--
4836 `isPropQuickApp f n` is an "approximate" predicate which returns `LBool.true`
4837   if `f` applied to `n` arguments is a proposition. -/
4838 private partial def isPropQuickApp : Expr → Nat → MetaM LBool
4839 | Expr.const c lvls _, arity => do let constType ← inferConstType c lvls; isArrowProp constType arity
4840 | Expr.fvar fvarId _,  arity => do let fvarType ← inferFVarType fvarId; isArrowProp fvarType arity
4841 | Expr.mvar mvarId _,  arity => do let mvarType ← inferMVarType mvarId; isArrowProp mvarType arity
4842 | Expr.app f _ _ ,      arity => isPropQuickApp f (arity+1)
4843 | Expr.mdata _ e _ ,      arity => isPropQuickApp e arity
4844 | Expr.letE _ _ _ b _ ,  arity => isPropQuickApp b arity
4845 | Expr.lam _ _ _ _ ,      0    => pure LBool.false
4846 | Expr.lam _ _ b _ ,     arity+1 => isPropQuickApp b arity
4847 | _,                    _    => pure LBool.undef
4848
4849 /--
4850 `isPropQuick e` is an "approximate" predicate which returns `LBool.true`
4851   if `e` is a proposition. -/
4852 partial def isPropQuick : Expr → MetaM LBool
4853 | Expr.bvar _ _      => pure LBool.undef
4854 | Expr.lit _ _       => pure LBool.false
4855 | Expr.sort _ _      => pure LBool.false
4856 | Expr.lam _ _ _ _   => pure LBool.false
4857 | Expr.letE _ _ _ b _ => isPropQuick b
4858 | Expr.proj _ _ _ _  => pure LBool.undef
4859 | Expr.forallE _ _ b _ => isPropQuick b
4860 | Expr.mdata _ e _   => isPropQuick e
4861 | Expr.const c lvls _ => do let constType ← inferConstType c lvls; isArrowProp constType 0
4862 | Expr.fvar fvarId _  => do let fvarType ← inferFVarType fvarId; isArrowProp fvarType 0
4863 | Expr.mvar mvarId _  => do let mvarType ← inferMVarType mvarId; isArrowProp mvarType 0
4864 | Expr.app f _ _     => isPropQuickApp f 1
4865
4866 /-- `isProp whnf e` return `true` if `e` is a proposition.
4867
4868   If `e` contains metavariables, it may not be possible
4869   to decide whether is a proposition or not. We return `false` in this
4870   case. We considered using `LBool` and retuning `LBool.undef`, but

```



```

4871     we have no applications for it. -/
4872 def isProp (e : Expr) : MetaM Bool := do
4873   let r ← isPropQuick e
4874   match r with
4875   | LBool.true  => pure true
4876   | LBool.false => pure false
4877   | LBool.undef =>
4878     let type ← inferType e
4879     let type ← whnfD type
4880     match type with
4881     | Expr.sort u _ => return isAlwaysZero (← instantiateLevelMVars u)
4882     | _             => pure false
4883
4884 /--
4885   `isArrowProposition type n` is an "approximate" predicate which returns `LBool.true`
4886   if `type` is of the form `A_1 -> ... -> A_n -> B`, where `B` is a proposition.
4887   Remark: `type` can be a dependent arrow. -/
4888 private partial def isArrowProposition : Expr → Nat → MetaM LBool
4889 | Expr.forallE _ _ b _, n+1 => isArrowProposition b n
4890 | Expr.letE _ _ _ b _, n   => isArrowProposition b n
4891 | Expr.mdata _ e _,       n   => isArrowProposition e n
4892 | type,                  0    => isPropQuick type
4893 | _,                    _    => pure LBool.undef
4894
4895 mutual
4896 /--
4897   `isProofQuickApp f n` is an "approximate" predicate which returns `LBool.true`
4898   if `f` applied to `n` arguments is a proof. -/
4899 private partial def isProofQuickApp : Expr → Nat → MetaM LBool
4900 | Expr.const c lvls _, arity => do let constType ← inferConstType c lvls; isArrowProposition constType arity
4901 | Expr.fvar fvarId _, arity => do let fvarType ← inferFVarType fvarId; isArrowProposition fvarType arity
4902 | Expr.mvar mvarId _, arity => do let mvarType ← inferMVarType mvarId; isArrowProposition mvarType arity
4903 | Expr.app f _ _, arity => isProofQuickApp f (arity+1)
4904 | Expr.mdata _ e _, arity => isProofQuickApp e arity
4905 | Expr.letE _ _ _ b _, arity => isProofQuickApp b arity
4906 | Expr.lam _ _ b _, 0 => isProofQuick b
4907 | Expr.lam _ _ b _, arity+1 => isProofQuickApp b arity
4908 | _, _ => pure LBool.undef
4909
4910 /--
4911   `isProofQuick e` is an "approximate" predicate which returns `LBool.true`
4912   if `e` is a proof. -/
4913 partial def isProofQuick : Expr → MetaM LBool
4914 | Expr.bvar _ _ => pure LBool.undef
4915 | Expr.lit _ _ => pure LBool.false
4916 | Expr.sort _ _ => pure LBool.false
4917 | Expr.lam _ _ b _ => isProofQuick b

```

```

4918 | Expr.letE _ _ _ b _ => isProofQuick b
4919 | Expr.proj _ _ _ _ => pure LBool.undef
4920 | Expr.forallE _ _ _ b _ => pure LBool.false
4921 | Expr.mdata _ e _ => isProofQuick e
4922 | Expr.const c lvls _ => do let constType ← inferConstType c lvls; isArrowProposition constType 0
4923 | Expr.fvar fvarId _ => do let fvarType ← inferFVarType fvarId; isArrowProposition fvarType 0
4924 | Expr.mvar mvarId _ => do let mvarType ← inferMVarType mvarId; isArrowProposition mvarType 0
4925 | Expr.app f _ _ => isProofQuickApp f 1
4926
4927 end
4928
4929 def isProof (e : Expr) : MetaM Bool := do
4930   let r ← isProofQuick e
4931   match r with
4932   | LBool.true => pure true
4933   | LBool.false => pure false
4934   | LBool.undef => do
4935     let type ← inferType e
4936     Meta.isProp type
4937
4938 /--
4939   `isArrowType type n` is an "approximate" predicate which returns `LBool.true`
4940   if `type` is of the form `A_1 -> ... -> A_n -> Sort _`.
4941   Remark: `type` can be a dependent arrow. -/
4942 private partial def isArrowType : Expr → Nat → MetaM LBool
4943 | Expr.sort u _, 0 => pure LBool.true
4944 | Expr.forallE _ _ _ _, 0 => pure LBool.false
4945 | Expr.forallE _ _ _ b _, n+1 => isArrowType b n
4946 | Expr.letE _ _ _ b _, n => isArrowType b n
4947 | Expr.mdata _ e _, n => isArrowType e n
4948 | _, _ => pure LBool.undef
4949
4950 /--
4951   `isTypeQuickApp f n` is an "approximate" predicate which returns `LBool.true`
4952   if `f` applied to `n` arguments is a type. -/
4953 private partial def isTypeQuickApp : Expr → Nat → MetaM LBool
4954 | Expr.const c lvls _, arity => do let constType ← inferConstType c lvls; isArrowType constType arity
4955 | Expr.fvar fvarId _, arity => do let fvarType ← inferFVarType fvarId; isArrowType fvarType arity
4956 | Expr.mvar mvarId _, arity => do let mvarType ← inferMVarType mvarId; isArrowType mvarType arity
4957 | Expr.app f _ _, arity => isTypeQuickApp f (arity+1)
4958 | Expr.mdata _ e _, arity => isTypeQuickApp e arity
4959 | Expr.letE _ _ _ b _, arity => isTypeQuickApp b arity
4960 | Expr.lam _ _ _ _, 0 => pure LBool.false
4961 | Expr.lam _ _ _ b _, arity+1 => isTypeQuickApp b arity
4962 | _, _ => pure LBool.undef
4963
4964 /--

```

```

4965 `isTypeQuick e` is an "approximate" predicate which returns `LBool.true`
4966 if `e` is a type. -/
4967 partial def isTypeQuick : Expr → MetaM LBool
4968 | Expr.bvar _ _      => pure LBool.undef
4969 | Expr.lit _ _       => pure LBool.false
4970 | Expr.sort _ _      => pure LBool.true
4971 | Expr.lam _ _ _ _    => pure LBool.false
4972 | Expr.letE _ _ _ b _ => isTypeQuick b
4973 | Expr.proj _ _ _ _   => pure LBool.undef
4974 | Expr.forallE _ _ b _ => pure LBool.true
4975 | Expr.mdata _ e _    => isTypeQuick e
4976 | Expr.const c lvls _ => do let constType ← inferConstType c lvls; isArrowType constType 0
4977 | Expr.fvar fvarId _  => do let fvarType ← inferFVarType fvarId; isArrowType fvarType 0
4978 | Expr.mvar mvarId _  => do let mvarType ← inferMVarType mvarId; isArrowType mvarType 0
4979 | Expr.app f _ _      => isTypeQuickApp f 1
4980
4981 def isType (e : Expr) : MetaM Bool := do
4982   let r ← isTypeQuick e
4983   match r with
4984   | LBool.true  => pure true
4985   | LBool.false => pure false
4986   | LBool.undef =>
4987     let type ← inferType e
4988     let type ← whnfD type
4989     match type with
4990     | Expr.sort _ _ => pure true
4991     | _              => pure false
4992
4993 partial def isTypeFormerType (type : Expr) : MetaM Bool := do
4994   let type ← whnfD type
4995   match type with
4996   | Expr.sort _ _ => pure true
4997   | Expr.forallE n d b c =>
4998     withLocalDecl' n c.binderInfo d fun fvar =>
4999       isTypeFormerType (b.instantiate1 fvar)
5000   | _ => pure false
5001
5002 /--
5003   Return true iff `e : Sort _` or `e : (forall As, Sort _)`.
5004   Remark: it subsumes `isType` -/
5005 def isTypeFormer (e : Expr) : MetaM Bool := do
5006   let type ← inferType e
5007   isTypeFormerType type
5008
5009 end Lean.Meta
5010 // ::::::::::::::
5011 // Instances.lean

```

```

5012 // ::::::::::::::
5013 /-
5014 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5015 Released under Apache 2.0 license as described in the file LICENSE.
5016 Authors: Leonardo de Moura
5017 -/
5018 import Lean.ScopedEnvExtension
5019 import Lean.Meta.DiscrTree
5020
5021 namespace Lean.Meta
5022
5023 structure InstanceEntry where
5024   keys      : Array DiscrTree.Key
5025   val       : Expr
5026   priority   : Nat
5027   globalName? : Option Name := none
5028   deriving Inhabited
5029
5030 instance : BEq InstanceEntry where
5031   beq e₁ e₂ := e₁.val == e₂.val
5032
5033 instance : ToFormat InstanceEntry where
5034   format e := match e.globalName? with
5035     | some n => fmt n
5036     | _      => "<local>"
5037
5038 structure Instances where
5039   discrTree      : DiscrTree InstanceEntry := DiscrTree.empty
5040   globalInstances : NameSet := {}
5041   deriving Inhabited
5042
5043 def addInstanceEntry (d : Instances) (e : InstanceEntry) : Instances := {
5044   d with
5045     discrTree := d.discrTree.insertCore e.keys e
5046     globalInstances := match e.globalName? with
5047       | some n => d.globalInstances.insert n
5048       | none   => d.globalInstances
5049 }
5050
5051 builtin_initialize instanceExtension : SimpleScopedEnvExtension InstanceEntry Instances ←
5052   registerSimpleScopedEnvExtension {
5053     name      := `instanceExt
5054     initial   := {}
5055     addEntry  := addInstanceEntry
5056   }
5057
5058 private def mkInstanceKey (e : Expr) : MetaM (Array DiscrTree.Key) := do

```

```

5059 let type ← inferType e
5060 withNewMCtxDepth do
5061   let (_, _, type) ← forallMetaTelescopeReducing type
5062   DiscrTree.mkPath type
5063
5064 def addInstance (declName : Name) (attrKind : AttributeKind) (prio : Nat) : MetaM Unit := do
5065   let cinfo ← getConstInfo declName
5066   let c := mkConst declName (cinfo.levelParams.map mkLevelParam)
5067   let keys ← mkInstanceKey c
5068   instanceExtension.add { keys := keys, val := c, priority := prio, globalName? := declName } attrKind
5069
5070 builtin_initialize
5071   registerBuiltinAttribute {
5072     name := `instance
5073     descr := "type class instance"
5074     add := fun declName stx attrKind => do
5075       let prio ← getAttrParamOptPrio stx[1]
5076       discard <| addInstance declName attrKind prio |>.run {} {}
5077   }
5078
5079 @[export lean_is_instance]
5080 def isGlobalInstance (env : Environment) (declName : Name) : Bool :=
5081   Meta.instanceExtension.getState env |>.globalInstances.contains declName
5082
5083 def getGlobalInstancesIndex : MetaM (DiscrTree InstanceEntry) :=
5084   return Meta.instanceExtension.getState (← getEnv) |>.discrTree
5085
5086 /- Default instance support -/
5087
5088 structure DefaultInstanceEntry where
5089   className      : Name
5090   instanceName   : Name
5091   priority       : Nat
5092
5093 abbrev PrioritySet := Std.RBTree Nat (.>.)
5094
5095 structure DefaultInstances where
5096   defaultInstances : NameMap (List (Name × Nat)) := {}
5097   priorities       : PrioritySet := {}
5098   deriving Inhabited
5099
5100 def addDefaultInstanceEntry (d : DefaultInstances) (e : DefaultInstanceEntry) : DefaultInstances :=
5101   let d := { d with priorities := d.priorities.insert e.priority }
5102   match d.defaultInstances.find? e.className with
5103   | some insts => { d with defaultInstances := d.defaultInstances.insert e.className <| (e.instanceName, e.priority) :: insts }
5104   | none      => { d with defaultInstances := d.defaultInstances.insert e.className [(e.instanceName, e.priority)] }
5105

```

```

5106 builtin_initialize defaultInstanceExtension : SimplePersistentEnvExtension DefaultInstanceEntry DefaultInstances ←
5107   registerSimplePersistentEnvExtension {
5108     name      := `defaultInstanceExt
5109     addEntryFn := addDefaultInstanceEntry
5110     addImportedFn := fun es => (mkStateFromImportedEntries addDefaultInstanceEntry {}) es)
5111   }
5112
5113 def addDefaultInstance (declName : Name) (prio : Nat := 0) : MetaM Unit := do
5114   match (← getEnv).find? declName with
5115   | none => throwError! "unknown constant '{declName}'"
5116   | some info =>
5117     forallTelescopeReducing info.type fun _ type => do
5118       match type.getAppFn with
5119       | Expr.const className _ _ =>
5120         unless isClass (← getEnv) className do
5121           throwError! "invalid default instance '{declName}', it has type '({className} ...)', but {className}' is not a type class"
5122         setEnv <| defaultInstanceExtension.addEntry (← getEnv) { className := className, instanceName := declName, priority := prio }
5123       | _ => throwError! "invalid default instance '{declName}', type must be of the form '(C ...)' where 'C' is a type class"
5124
5125 builtin_initialize
5126   registerBuiltinAttribute {
5127     name      := `defaultInstance
5128     descr := "type class default instance"
5129     add      := fun declName stx kind => do
5130       let prio ← getAttrParamOptPrio stx[1]
5131       unless kind == AttributeKind.global do throwError "invalid attribute 'defaultInstance', must be global"
5132       discard <| addDefaultInstance declName prio |>.run {} {}
5133   }
5134
5135 def getDefaultInstancesPriorities [Monad m] [MonadEnv m] : m PrioritySet :=
5136   return defaultInstanceExtension.getState (← getEnv) |>.priorities
5137
5138 def getDefaultInstances [Monad m] [MonadEnv m] (className : Name) : m (List (Name × Nat)) :=
5139   return defaultInstanceExtension.getState (← getEnv) |>.defaultInstances.find? className |>.getD []
5140
5141 end Lean.Meta
5142 // ::::::::::::::
5143 // KAbstract.lean
5144 // ::::::::::::::
5145 /-
5146 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
5147 Released under Apache 2.0 license as described in the file LICENSE.
5148 Author: Leonardo de Moura
5149 -/
5150 import Lean.Data.Occurrences
5151 import Lean.HeadIndex
5152 import Lean.Meta.ExprDefEq

```

```

5153
5154 namespace Lean.Meta
5155
5156 def kabstract (e : Expr) (p : Expr) (occs : Occurrences := Occurrences.all) : MetaM Expr := do
5157   let e ← instantiateMVars e
5158   if p.isFVar && occs == Occurrences.all then
5159     return e.abstract #[p] -- Easy case
5160   else
5161     let pHeadIdx := p.toHeadIndex
5162     let pNumArgs := p.headNumArgs
5163     let rec visit (e : Expr) (offset : Nat) : StateRefT Nat MetaM Expr := do
5164       let visitChildren : Unit → StateRefT Nat MetaM Expr := fun _ => do
5165         match e with
5166         | Expr.app f a _ => return e.updateApp! (← visit f offset) (← visit a offset)
5167         | Expr.mdata _ b _ => return e.updateMData! (← visit b offset)
5168         | Expr.proj _ _ b _ => return e.updateProj! (← visit b offset)
5169         | Expr.letE _ t v b _ => return e.updateLet! (← visit t offset) (← visit v offset) (← visit b (offset+1))
5170         | Expr.lam _ d b _ => return e.updateLambdaE! (← visit d offset) (← visit b (offset+1))
5171         | Expr.forallE _ d b _ => return e.updateForallE! (← visit d offset) (← visit b (offset+1))
5172         | e => return e
5173     if e.hasLooseBVars then
5174       visitChildren ()
5175     else if e.toHeadIndex != pHeadIdx || e.headNumArgs != pNumArgs then
5176       visitChildren ()
5177     else if (← isDefEq e p) then
5178       let i ← get
5179       set (i+1)
5180       if occs.contains i then
5181         pure (mkBVar offset)
5182       else
5183         visitChildren ()
5184     else
5185       visitChildren ()
5186   visit e 0 |>.run' 1
5187
5188 end Lean.Meta
5189 // ::::::::::::::
5190 // LevelDefEq.lean
5191 // ::::::::::::::
5192 /-
5193 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5194 Released under Apache 2.0 license as described in the file LICENSE.
5195 Authors: Leonardo de Moura
5196 -/
5197 import Lean.Meta.Basic
5198 import Lean.Meta.InferType
5199

```

```

5200 namespace Lean.Meta
5201
5202 private partial def decAux? : Level → MetaM (Option Level)
5203 | Level.zero _      => return none
5204 | Level.param _ _   => return none
5205 | Level.mvar mvarId _ => do
5206   let mctx ← getMCtx
5207   match mctx.getLevelAssignment? mvarId with
5208   | some u => decAux? u
5209   | none   =>
5210     if (← isReadOnlyLevelMVar mvarId) then
5211       return none
5212     else
5213       let u ← mkFreshLevelMVar
5214       assignLevelMVar mvarId (mkLevelSucc u)
5215       return u
5216 | Level.succ u _ => return u
5217 | u =>
5218   let process (u v : Level) : MetaM (Option Level) := do
5219     match (← decAux? u) with
5220     | none   => return none
5221     | some u => do
5222       match (← decAux? v) with
5223       | none   => return none
5224       | some v => return mkLevelMax' u v
5225   match u with
5226   | Level.max u v _ => process u v
5227   /- Remark: If `decAux? v` returns `some ...`, then `imax u v` is equivalent to `max u v`. -/
5228   | Level.imax u v _ => process u v
5229   | _                => unreachable!
5230
5231 def decLevel? (u : Level) : MetaM (Option Level) := do
5232   let mctx ← getMCtx
5233   match (← decAux? u) with
5234   | some v => return some v
5235   | none   => do
5236     modify fun s => { s with mctx := mctx }
5237     return none
5238
5239 def decLevel (u : Level) : MetaM Level := do
5240   match (← decLevel? u) with
5241   | some u => return u
5242   | none   => throwError! "invalid universe level, {u} is not greater than 0"
5243
5244 /- This method is useful for inferring universe level parameters for function that take arguments such as `{α : Type u}`.
5245 Recall that `Type u` is `Sort (u+1)` in Lean. Thus, given `α`, we must infer its universe level,
5246 and then decrement 1 to obtain `u`. -/

```



```

5247 def getDecLevel (type : Expr) : MetaM Level := do
5248   decLevel (← getLevel type)
5249
5250 /--
5251   Return true iff `lvl` occurs in `max u_1 ... u_n` and `lvl != u_i` for all `i in [1, n]`.
5252   That is, `lvl` is a proper level subterm of some `u_i`. -/
5253 private def strictOccursMax (lvl : Level) : Level → Bool
5254   | Level.max u v _ => visit u || visit v
5255   | _               => false
5256 where
5257   visit : Level → Bool
5258   | Level.max u v _ => visit u || visit v
5259   | u               => u != lvl && lvl.occurs u
5260
5261 /-- `mkMaxArgsDiff mvarId (max u_1 ... (mvar mvarId) ... u_n) v` => `max v u_1 ... u_n` -/
5262 private def mkMaxArgsDiff (mvarId : MVarId) : Level → Level → Level
5263   | Level.max u v _, acc => mkMaxArgsDiff mvarId v <| mkMaxArgsDiff mvarId u acc
5264   | l@(Level.mvar id _), acc => if id != mvarId then mkLevelMax' acc l else acc
5265   | l, acc => mkLevelMax' acc l
5266
5267 /--
5268   Solve `?m == max ?m v` by creating a fresh metavariable `?n`
5269   and assigning `?m := max ?n v` -/
5270 private def solveSelfMax (mvarId : MVarId) (v : Level) : MetaM Unit := do
5271   assert! v.isMax
5272   let n ← mkFreshLevelMVar
5273   assignLevelMVar mvarId <| mkMaxArgsDiff mvarId v n
5274
5275 private def postponeIsLevelDefEq (lhs : Level) (rhs : Level) : MetaM Unit :=
5276   modifyPostponed fun postponed => postponed.push { lhs := lhs, rhs := rhs }
5277
5278 mutual
5279
5280 private partial def solve (u v : Level) : MetaM LBool := do
5281   match u, v with
5282   | Level.mvar mvarId _, _ =>
5283     if (← isReadOnlyLevelMVar mvarId) then
5284       return LBool.undef
5285     else if !u.occurs v then
5286       assignLevelMVar u.mvarId! v
5287       return LBool.true
5288     else if v.isMax && !strictOccursMax u v then
5289       solveSelfMax u.mvarId! v
5290       return LBool.true
5291     else
5292       return LBool.undef
5293   | Level.zero _, Level.max v1 v2 _ =>

```

```

5294   Bool.toLBool <$> (isLevelDefEqAux levelZero v₁ <&&> isLevelDefEqAux levelZero v₂)
5295 | Level.zero _, Level.imax _ v₂ _ =>
5296   Bool.toLBool <$> isLevelDefEqAux levelZero v₂
5297 | Level.zero _, Level.succ .. => return LBool.false
5298 | Level.succ u _, v =>
5299   if u.isMVar && u.occurs v then
5300     return LBool.undef
5301   else
5302     match (← Meta.decLevel? v) with
5303     | some v => Bool.toLBool <$> isLevelDefEqAux u v
5304     | none   => return LBool.undef
5305 | _, _ => return LBool.undef
5306
5307 partial def isLevelDefEqAux : Level → Level → MetaM Bool
5308 | Level.succ lhs _, Level.succ rhs _ => isLevelDefEqAux lhs rhs
5309 | lhs, rhs => do
5310   if lhs == rhs then
5311     return true
5312   else
5313     trace[Meta.isLevelDefEq.step]! "{lhs} =?= {rhs}"
5314     let lhs' ← instantiateLevelMVars lhs
5315     let lhs' := lhs'.normalize
5316     let rhs' ← instantiateLevelMVars rhs
5317     let rhs' := rhs'.normalize
5318     if lhs != lhs' || rhs != rhs' then
5319       isLevelDefEqAux lhs' rhs'
5320     else
5321       let r ← solve lhs rhs;
5322       if r != LBool.undef then
5323         return r == LBool.true
5324       else
5325         let r ← solve rhs lhs;
5326         if r != LBool.undef then
5327           return r == LBool.true
5328         else do
5329           let mctx ← getMCtx
5330           if !mctx.hasAssignableLevelMVar lhs && !mctx.hasAssignableLevelMVar rhs then
5331             let ctx ← read
5332             if ctx.config.isDefEqStuckEx && (lhs.isMVar || rhs.isMVar) then do
5333               trace[Meta.isLevelDefEq.stuck]! "{lhs} =?= {rhs}"
5334               Meta.throwIsDefEqStuck
5335             else
5336               return false
5337         else
5338           postponeIsLevelDefEq lhs rhs
5339       return true
5340 end

```

```

5341
5342 def isListLevelDefEqAux : List Level → List Level → MetaM Bool
5343 | [], [] => return true
5344 | u::us, v::vs => isLevelDefEqAux u v <&&> isListLevelDefEqAux us vs
5345 | _, _ => return false
5346
5347 private def getNumPostponed : MetaM Nat := do
5348   return (← getPostponed).size
5349
5350 open Std (PersistentArray)
5351
5352 private def getResetPostponed : MetaM (PersistentArray PostponedEntry) := do
5353   let ps ← getPostponed
5354   setPostponed {}
5355   return ps
5356
5357 private def processPostponedStep : MetaM Bool :=
5358   traceCtx `Meta.isLevelDefEq.postponed.step do
5359     let ps ← getResetPostponed
5360     for p in ps do
5361       unless (← isLevelDefEqAux p.lhs p.rhs) do
5362         return false
5363     return true
5364
5365 private partial def processPostponed (mayPostpone : Bool := true) : MetaM Bool := do
5366   if (← getNumPostponed) == 0 then
5367     return true
5368   else
5369     traceCtx `Meta.isLevelDefEq.postponed do
5370       let rec loop : MetaM Bool := do
5371         let numPostponed ← getNumPostponed
5372         if numPostponed == 0 then
5373           return true
5374         else
5375           trace[Meta.isLevelDefEq.postponed]! "processing #{numPostponed} postponed is-def-eq level constraints"
5376           if !(← processPostponedStep) then
5377             return false
5378           else
5379             let numPostponed' ← getNumPostponed
5380             if numPostponed' == 0 then
5381               return true
5382             else if numPostponed' < numPostponed then
5383               loop
5384             else
5385               trace[Meta.isLevelDefEq.postponed]! "no progress solving pending is-def-eq level constraints"
5386               return mayPostpone
5387       loop

```

```

5388
5389 private def restore (env : Environment) (mctx : MetavarContext) (postponed : PersistentArray PostponedEntry) : MetaM Unit := do
5390   setEnv env
5391   setMCtx mctx
5392   setPostponed postponed
5393
5394 /--
5395   `commitWhen x` executes `x` and process all postponed universe level constraints produced by `x`.
5396   We keep the modifications only if `processPostponed` return true and `x` returned `true`.
5397
5398   Remark: postponed universe level constraints must be solved before returning. Otherwise,
5399   we don't know whether `x` really succeeded. -/
5400 @[specialize] def commitWhen (x : MetaM Bool) (mayPostpone : Bool := true) : MetaM Bool := do
5401   let env ← getEnv
5402   let mctx ← getMCtx
5403   let postponed ← getResetPostponed
5404   try
5405     if (← x) then
5406       if (← processPostponed mayPostpone) then
5407         return true
5408       else
5409         restore env mctx postponed
5410         return false
5411     else
5412       restore env mctx postponed
5413       return false
5414   catch ex =>
5415     restore env mctx postponed
5416     throw ex
5417
5418 private def postponedToMessageData (ps : PersistentArray PostponedEntry) : MessageData := do
5419   let mut r := MessageData.nil
5420   for p in ps do
5421     r := m! "{r}\n{p.lhs} =?= {p.rhs}"
5422   return r
5423
5424 @[specialize] def withoutPostponingUniverseConstraintsImp {α} (x : MetaM α) : MetaM α := do
5425   let postponed ← getResetPostponed
5426   try
5427     let a ← x
5428     unless (← processPostponed (mayPostpone := false)) do
5429       throwError! "stuck at solving universe constraints{MessageData.nestD (postponedToMessageData (← getPostponed))}"
5430     setPostponed postponed
5431     return a
5432   catch ex =>
5433     setPostponed postponed
5434     throw ex

```

```

5435
5436 @[inline] def withoutPostponingUniverseConstraints {α m} [MonadControlT MetaM m] [Monad m] : m α → m α :=
5437   mapMetaM <| withoutPostponingUniverseConstraintsImp
5438
5439 def isLevelDefEq (u v : Level) : MetaM Bool :=
5440   traceCtx `Meta.isLevelDefEq do
5441     let b ← commitWhen (mayPostpone := true) <| Meta.isLevelDefEqAux u v
5442     trace[Meta.isLevelDefEq]! "{u} =?= {v} ... {if b then \"success\" else \"failure\"}"
5443     return b
5444
5445 def isExprDefEq (t s : Expr) : MetaM Bool :=
5446   traceCtx `Meta.isDefEq do
5447     let b ← commitWhen (mayPostpone := true) <| Meta.isExprDefEqAux t s
5448     trace[Meta.isDefEq]! "{t} =?= {s} ... {if b then \"success\" else \"failure\"}"
5449     return b
5450
5451 abbrev isDefEq (t s : Expr) : MetaM Bool :=
5452   isExprDefEq t s
5453
5454 def isExprDefEqGuarded (a b : Expr) : MetaM Bool := do
5455   try isExprDefEq a b catch _ => return false
5456
5457 abbrev isDefEqGuarded (t s : Expr) : MetaM Bool :=
5458   isExprDefEqGuarded t s
5459
5460 def isDefEqNoConstantApprox (t s : Expr) : MetaM Bool :=
5461   approxDefEq <| isDefEq t s
5462
5463 builtin_initialize
5464   registerTraceClass `Meta.isLevelDefEq
5465   registerTraceClass `Meta.isLevelDefEq.step
5466   registerTraceClass `Meta.isLevelDefEq.postponed
5467
5468 end Lean.Meta
5469 // ::::::::::::::
5470 // Match.lean
5471 // ::::::::::::::
5472 /-
5473 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5474 Released under Apache 2.0 license as described in the file LICENSE.
5475 Authors: Leonardo de Moura
5476 -/
5477 import Lean.Meta.Match.MatchPatternAttr
5478 import Lean.Meta.Match.Match
5479 import Lean.Meta.Match.CaseValues
5480 import Lean.Meta.Match.CaseArraySizes
5481

```

```

5482 namespace Lean
5483
5484 builtin_initialize registerTraceClass `Meta.Match
5485
5486 end Lean
5487 // ::::::::::::::
5488 // MatchUtil.lean
5489 // ::::::::::::::
5490 /-
5491 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
5492 Released under Apache 2.0 license as described in the file LICENSE.
5493 Authors: Leonardo de Moura
5494 -/
5495 import Lean.Util.Recognizers
5496 import Lean.Meta.Basic
5497
5498 namespace Lean.Meta
5499
5500 @[inline] def testHelper (e : Expr) (p : Expr → MetaM Bool) : MetaM Bool := do
5501   if (← p e) then
5502     return true
5503   else
5504     p (← whnf e)
5505
5506 @[inline] def matchHelper? (e : Expr) (p? : Expr → MetaM (Option α)) : MetaM (Option α) := do
5507   match (← p? e) with
5508   | none => p? (← whnf e)
5509   | s    => return s
5510
5511 def matchEq? (e : Expr) : MetaM (Option (Expr × Expr × Expr)) :=
5512   matchHelper? e fun e => return Expr.eq? e
5513
5514 def matchFalse (e : Expr) : MetaM Bool := do
5515   testHelper e fun e => return e.isConstOf ``False
5516
5517 def matchNot? (e : Expr) : MetaM (Option Expr) :=
5518   matchHelper? e fun e => do
5519     if let some e := e.not? then
5520       return e
5521     else if let some (a, b) := e.arrow? then
5522       if (← matchFalse b) then return some a else return none
5523     else
5524       return none
5525
5526 def matchNe? (e : Expr) : MetaM (Option (Expr × Expr × Expr)) :=
5527   matchHelper? e fun e => do
5528     if let some r := e.ne? then

```

```

5529     return r
5530   else if let some e ← matchNot? e then
5531     matchEq? e
5532   else
5533     return none
5534
5535 def matchConstructorApp? (e : Expr) : MetaM (Option ConstructorVal) := do
5536   let env ← getEnv
5537   matchHelper? e fun e =>
5538     return e.isConstructorApp? env
5539
5540 end Lean.Meta
5541 // ::::::::::::::
5542 // Offset.lean
5543 // ::::::::::::::
5544 /-
5545 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
5546 Released under Apache 2.0 license as described in the file LICENSE.
5547 Authors: Leonardo de Moura
5548 -/
5549 import Lean.Data.LBool
5550 import Lean.Meta.InferType
5551
5552 namespace Lean.Meta
5553
5554 private abbrev withInstantiatedMVars (e : Expr) (k : Expr → OptionT MetaM  $\alpha$ ) : OptionT MetaM  $\alpha$  := do
5555   let eNew ← instantiateMVars e
5556   if eNew.getAppFn.isMVar then
5557     failure
5558   else
5559     k eNew
5560
5561 /-
5562 Evaluate simple `Nat` expressions.
5563 Remark: this method assumes the given expression has type `Nat`. -/
5564 partial def evalNat : Expr → OptionT MetaM Nat
5565 | Expr.lit (Literal.natVal n) _ => return n
5566 | Expr.mdata _ e _             => evalNat e
5567 | Expr.const `Nat.zero ..      => return 0
5568 | e@(Expr.app ..)              => visit e
5569 | e@(Expr.mvar ..)             => visit e
5570 | _                             => failure
5571 where
5572   visit e := do
5573     let f := e.getAppFn
5574     match f with
5575     | Expr.mvar .. => withInstantiatedMVars e evalNat

```

```

5576 | Expr.const c _ _ =>
5577   let nargs := e.getAppNumArgs
5578   if c == ``Nat.succ && nargs == 1 then
5579     let v ← evalNat (e.getArg! 0)
5580     return v+1
5581   else if c == ``Nat.add && nargs == 2 then
5582     let v1 ← evalNat (e.getArg! 0)
5583     let v2 ← evalNat (e.getArg! 1)
5584     return v1 + v2
5585   else if c == ``Nat.sub && nargs == 2 then
5586     let v1 ← evalNat (e.getArg! 0)
5587     let v2 ← evalNat (e.getArg! 1)
5588     return v1 - v2
5589   else if c == ``Nat.mul && nargs == 2 then
5590     let v1 ← evalNat (e.getArg! 0)
5591     let v2 ← evalNat (e.getArg! 1)
5592     return v1 * v2
5593   else if c == ``Add.add && nargs == 4 then
5594     let v1 ← evalNat (e.getArg! 2)
5595     let v2 ← evalNat (e.getArg! 3)
5596     return v1 + v2
5597   else if c == ``Sub.sub && nargs == 4 then
5598     let v1 ← evalNat (e.getArg! 2)
5599     let v2 ← evalNat (e.getArg! 3)
5600     return v1 - v2
5601   else if c == ``Mul.mul && nargs == 4 then
5602     let v1 ← evalNat (e.getArg! 2)
5603     let v2 ← evalNat (e.getArg! 3)
5604     return v1 * v2
5605   else if c == ``HAdd.hAdd && nargs == 6 then
5606     let v1 ← evalNat (e.getArg! 3)
5607     let v2 ← evalNat (e.getArg! 5)
5608     return v1 + v2
5609   else if c == ``HSub.hSub && nargs == 6 then
5610     let v1 ← evalNat (e.getArg! 4)
5611     let v2 ← evalNat (e.getArg! 5)
5612     return v1 - v2
5613   else if c == ``HMul.hMul && nargs == 6 then
5614     let v1 ← evalNat (e.getArg! 4)
5615     let v2 ← evalNat (e.getArg! 5)
5616     return v1 * v2
5617   else if c == ``OfNat.ofNat && nargs == 3 then
5618     evalNat (e.getArg! 1)
5619   else
5620     failure
5621 | _ => failure
5622

```



```

5623 /- Quick function for converting `e` into `s + k` s.t. `e` is definitionally equal to `Nat.add s k`. -/
5624 private partial def getOffsetAux : Expr → Bool → OptionT MetaM (Expr × Nat)
5625 | e@(Expr.app _ a _) , top => do
5626   let f := e.getAppFn
5627   match f with
5628   | Expr.mvar .. => withInstantiatedMVars e (getOffsetAux · top)
5629   | Expr.const c _ =>
5630     let nargs := e.getAppNumArgs
5631     if c == ``Nat.succ && nargs == 1 then do
5632       let (s, k) ← getOffsetAux a false
5633       pure (s, k+1)
5634     else if c == ``Nat.add && nargs == 2 then do
5635       let v ← evalNat (e.getArg! 1)
5636       let (s, k) ← getOffsetAux (e.getArg! 0) false
5637       pure (s, k+v)
5638     else if c == ``Add.add && nargs == 4 then do
5639       let v ← evalNat (e.getArg! 3)
5640       let (s, k) ← getOffsetAux (e.getArg! 2) false
5641       pure (s, k+v)
5642     else if c == ``HAdd.hAdd && nargs == 6 then do
5643       let v ← evalNat (e.getArg! 5)
5644       let (s, k) ← getOffsetAux (e.getArg! 4) false
5645       pure (s, k+v)
5646     else if top then failure else pure (e, 0)
5647 | _ => if top then failure else pure (e, 0)
5648 | e, top => if top then failure else pure (e, 0)
5649
5650 private def getOffset (e : Expr) : OptionT MetaM (Expr × Nat) :=
5651   getOffsetAux e true
5652
5653 private partial def isOffset : Expr → OptionT MetaM (Expr × Nat)
5654 | e@(Expr.app _ a _) =>
5655   let f := e.getAppFn
5656   match f with
5657   | Expr.mvar .. => withInstantiatedMVars e isOffset
5658   | Expr.const c _ =>
5659     let nargs := e.getAppNumArgs
5660     if (c == ``Nat.succ && nargs == 1) || (c == ``Nat.add && nargs == 2) || (c == ``Add.add && nargs == 4) || (c == ``HAdd.hAdd &&
nargs == 6) then
5661       getOffset e
5662     else
5663       failure
5664 | _ => failure
5665 | _ => failure
5666
5667 private def isNatZero (e : Expr) : MetaM Bool := do
5668   match (← evalNat e) with

```

```

5669 | some v => v == 0
5670 | _      => false
5671
5672 private def mkOffset (e : Expr) (offset : Nat) : MetaM Expr := do
5673   if offset == 0 then
5674     return e
5675   else if (← isNatZero e) then
5676     return mkNatLit offset
5677   else
5678     return mkAppB (mkConst ``Nat.add) e (mkNatLit offset)
5679
5680 def isDefEqOffset (s t : Expr) : MetaM LBool := do
5681   let ifNatExpr (x : MetaM LBool) : MetaM LBool := do
5682     let type ← inferType s
5683     -- Remark: we use `withNewMCtxDepth` to make sure we don't assing metavariables when performing the `isDefEq` test
5684     if (← withNewMCtxDepth <| Meta.isExprDefEqAux type (mkConst ``Nat)) then
5685       x
5686     else
5687       return LBool.undef
5688   let isDefEq (s t) : MetaM LBool :=
5689     ifNatExpr <| toLBoolM <| Meta.isExprDefEqAux s t
5690   match (← isOffset s) with
5691   | some (s, k1) =>
5692     match (← isOffset t) with
5693     | some (t, k2) => -- s+k1 ==?= t+k2
5694       if k1 == k2 then
5695         isDefEq s t
5696       else if k1 < k2 then
5697         isDefEq s (← mkOffset t (k2 - k1))
5698       else
5699         isDefEq (← mkOffset s (k1 - k2)) t
5700   | none =>
5701     match (← evalNat t) with
5702     | some v2 => -- s+k1 ==?= v2
5703       if v2 ≥ k1 then
5704         isDefEq s (mkNatLit $ v2 - k1)
5705       else
5706         ifNatExpr <| return LBool.false
5707   | none =>
5708     return LBool.undef
5709 | none =>
5710   match (← evalNat s) with
5711   | some v1 =>
5712     match (← isOffset t) with
5713     | some (t, k2) => -- v1 ==?= t+k2
5714       if v1 ≥ k2 then
5715         isDefEq (mkNatLit $ v1 - k2) t

```

```

5716         else
5717             ifNatExpr <| return LBool.false
5718         | none =>
5719             match (← evalNat t) with
5720             | some v₂ => ifNatExpr <| return (v₁ == v₂).toLBool -- v₁ == v₂
5721             | none   => return LBool.undef
5722         | none => return LBool.undef
5723
5724 end Lean.Meta
5725 // ::::::::::::::
5726 // PPGoal.lean
5727 // ::::::::::::::
5728 /-
5729 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
5730 Released under Apache 2.0 license as described in the file LICENSE.
5731 Author: Leonardo de Moura
5732 -/
5733 import Lean.Hygiene
5734 import Lean.Meta.InferType
5735
5736 namespace Lean.Meta
5737
5738 def ppAuxDeclsDefault := false
5739 builtin_initialize
5740   registerOption `pp.auxDecls { defValue := ppAuxDeclsDefault, group := "pp", descr := "display auxiliary declarations used to compile
recursive functions" }
5741 def getAuxDeclsOption (o : Options) : Bool :=
5742   o.get `pp.auxDecls ppAuxDeclsDefault
5743
5744 def ppInaccessibleNamesDefault := false
5745 builtin_initialize
5746   registerOption `pp.inaccessibleNames { defValue := ppInaccessibleNamesDefault, group := "pp", descr := "display inaccessible
declarations in the local context" }
5747 def getInaccessibleNamesOption (o : Options) : Bool :=
5748   o.get `pp.inaccessibleNames ppInaccessibleNamesDefault
5749
5750 namespace ToHide
5751
5752 structure State where
5753   hiddenInaccessibleProp : NameSet := {} -- FVarIds of Propostions with inaccessible names but containing only visible names. We show
only their types
5754   hiddenInaccessible      : NameSet := {} -- FVarIds with inaccessible names, but not in hiddenInaccessibleProp
5755   modified                : Bool := false
5756
5757 structure Context where
5758   goalTarget : Expr
5759

```

```

5760 abbrev M := ReaderT Context $ StateRefT State MetaM
5761
5762 /- Return true if `fvarId` is marked as an hidden inaccessible or inaccessible proposition -/
5763 def isMarked (fvarId : FVarId) : M Bool := do
5764   let s ← get
5765   return s.hiddenInaccessible.contains fvarId || s.hiddenInaccessibleProp.contains fvarId
5766
5767 /- If `fvarId` isMarked, then unmark it. -/
5768 def unmark (fvarId : FVarId) : M Unit := do
5769   modify fun s => {
5770     hiddenInaccessible      := s.hiddenInaccessible.erase fvarId
5771     hiddenInaccessibleProp := s.hiddenInaccessibleProp.erase fvarId
5772     modified                := true
5773   }
5774
5775 def moveToHiddeProp (fvarId : FVarId) : M Unit := do
5776   modify fun s => {
5777     hiddenInaccessible      := s.hiddenInaccessible.erase fvarId
5778     hiddenInaccessibleProp := s.hiddenInaccessibleProp.insert fvarId
5779     modified                := true
5780   }
5781
5782
5783 /- Return true if the given local declaration has a "visible dependency", that is, it contains
5784   a free variable that is `hiddenInaccessible`
5785
5786   Recall that hiddenInaccessibleProps are visible, only their names are hidden -/
5787 def hasVisibleDep (localDecl : LocalDecl) : M Bool := do
5788   let s ← get
5789   return (← getMCtx).findLocalDeclDependsOn localDecl fun fvarId =>
5790     !s.hiddenInaccessible.contains fvarId
5791
5792 /- Return true if the given local declaration has a "nonvisible dependency", that is, it contains
5793   a free variable that is `hiddenInaccessible` or `hiddenInaccessibleProp` -/
5794 def hasInaccessibleNameDep (localDecl : LocalDecl) : M Bool := do
5795   let s ← get
5796   return (← getMCtx).findLocalDeclDependsOn localDecl fun fvarId =>
5797     s.hiddenInaccessible.contains fvarId || s.hiddenInaccessibleProp.contains fvarId
5798
5799 /- If `e` is visible, then any inaccessible in `e` marked as hidden should be unmarked. -/
5800 partial def visitVisibleExpr (e : Expr) : M Unit := do
5801   visit (← instantiateMVars e) |>.run
5802 where
5803   visit (e : Expr) : MonadCacheT Expr Unit M Unit := do
5804     if e.hasFVar then
5805       checkCache e fun _ => do
5806         match e with

```

```

5807 | Expr.forallE _ d b _ => visit d; visit b
5808 | Expr.lam _ d b _ => visit d; visit b
5809 | Expr.letE _ t v b _ => visit t; visit v; visit b
5810 | Expr.app f a _ => visit f; visit a
5811 | Expr.mdata _ b _ => visit b
5812 | Expr.proj _ _ b _ => visit b
5813 | Expr.fvar fvarId _ => if (← isMarked fvarId) then unmark fvarId
5814 | _ => pure ()
5815
5816 def fixpointStep : M Unit := do
5817   visitVisibleExpr (← read).goalTarget -- The goal target is a visible forward dependency
5818   (← getLCtx).forM fun localDecl => do
5819     let fvarId := localDecl.fvarId
5820     if (← get).hiddenInaccessible.contains fvarId then
5821       if (← hasVisibleDep localDecl) then
5822         /- localDecl is marked to be hidden, but it has a (backward) visible dependency. -/
5823         unmark fvarId
5824       if (← isProp localDecl.type) then
5825         unless (← hasInaccessibleNameDep localDecl) do
5826           moveToHiddeProp fvarId
5827     else
5828       visitVisibleExpr localDecl.type
5829       match localDecl.value? with
5830       | some value => visitVisibleExpr value
5831       | _ => pure ()
5832
5833 partial def fixpoint : M Unit := do
5834   modify fun s => { s with modified := false }
5835   fixpointStep
5836   if (← get).modified then
5837     fixpoint
5838
5839 /-
5840 If pp.inaccessibleNames == false, then collect two sets of `FVarId`s : `hiddenInaccessible` and `hiddenInaccessibleProp`
5841 1- `hiddenInaccessible` contains `FVarId`s of free variables with inaccessible names that
5842    a) are not propositions or are propositions containing "visible" names.
5843 2- `hiddenInaccessibleProp` contains `FVarId`s of free variables with inaccessible names that are propositions
5844    containing "visible" names.
5845 Both sets do not contain `FVarId`s that contain visible backward or forward dependencies.
5846 The `goalTarget` counts as a forward dependency.
5847
5848 We say a name is visible if it is a free variable with FVarId not in `hiddenInaccessible` nor `hiddenInaccessibleProp`
5849 -/
5850 def collect (goalTarget : Expr) : MetaM (NameSet × NameSet) := do
5851   if getInaccessibleNamesOption (← getOptions) then
5852     /- Don't hide inaccessible names when `pp.inaccessibleNames` is set to true. -/
5853     return ({}, {})

```

```

5854 else
5855   let lctx ← getLCtx
5856   let hiddenInaccessible := lctx.foldl (init := {}) fun hiddenInaccessible localDecl => do
5857     if isInaccessibleUserName localDecl.userName then
5858       hiddenInaccessible.insert localDecl.fvarId
5859     else
5860       hiddenInaccessible
5861   let (_, s) ← fixpoint.run { goalTarget := goalTarget } |>.run { hiddenInaccessible := hiddenInaccessible }
5862   return (s.hiddenInaccessible, s.hiddenInaccessibleProp)
5863
5864 end ToHide
5865
5866 private def addLine (fmt : Format) : Format :=
5867   if fmt.isNil then fmt else fmt ++ Format.line
5868
5869 def ppGoal (mvarId : MVarId) : MetaM Format := do
5870   match (← getMCtx).findDecl? mvarId with
5871   | none      => pure "unknown goal"
5872   | some mvarDecl => do
5873     let indent      := 2 -- Use option
5874     let ppAuxDecls  := getAuxDeclsOption (← getOptions)
5875     let lctx        := mvarDecl.lctx
5876     let lctx        := lctx.sanitizeNames.run' { options := (← getOptions) }
5877     withLCtx lctx mvarDecl.localInstances do
5878       let (hidden, hiddenProp) ← ToHide.collect mvarDecl.type
5879       -- The following two `let rec`s are being used to control the generated code size.
5880       -- Then should be remove after we rewrite the compiler in Lean
5881       let rec pushPending (ids : List Name) (type? : Option Expr) (fmt : Format) : MetaM Format :=
5882         if ids.isEmpty then
5883           pure fmt
5884         else
5885           let fmt := addLine fmt
5886           match ids, type? with
5887           | [], _      => pure fmt
5888           | _, none    => pure fmt
5889           | _, some type => do
5890             let typeFmt ← ppExpr type
5891             pure $ fmt ++ (Format.joinSep ids.reverse " " ++ " :" ++ Format.nest indent (Format.line ++ typeFmt)).group
5892       let rec ppVars (varNames : List Name) (prevType? : Option Expr) (fmt : Format) (localDecl : LocalDecl) : MetaM (List Name ×
Option Expr × Format) := do
5893         if hiddenProp.contains localDecl.fvarId then
5894           let fmt ← pushPending varNames prevType? fmt
5895           let fmt := addLine fmt
5896           let type ← instantiateMVars localDecl.type
5897           let typeFmt ← ppExpr type
5898           let fmt := fmt ++ " :" ++ typeFmt
5899           pure ([], none, fmt)

```

```

5900     else
5901         match localDecl with
5902         | LocalDecl.cdecl _ _ varName type _ =>
5903             let varName := varName.simpMacroScopes
5904             let type ← instantiateMVars type
5905             if prevType? == none || prevType? == some type then
5906                 pure (varName :: varNames, some type, fmt)
5907             else do
5908                 let fmt ← pushPending varNames prevType? fmt
5909                 pure ([varName], some type, fmt)
5910         | LocalDecl.ldecl _ _ varName type val _ => do
5911             let varName := varName.simpMacroScopes
5912             let fmt ← pushPending varNames prevType? fmt
5913             let fmt := addLine fmt
5914             let type ← instantiateMVars type
5915             let val ← instantiateMVars val
5916             let typeFmt ← ppExpr type
5917             let valFmt ← ppExpr val
5918             let fmt := fmt ++ (format varName ++ " : " ++ typeFmt ++ " := " ++ Format.nest indent (Format.line ++ valFmt)).group
5919             pure ([], none, fmt)
5920 let (varNames, type?, fmt) ← lctx.foldlM (init := ([], none, Format.nil)) fun (varNames, prevType?, fmt) (localDecl : LocalDecl)
5921 =>
5922     if !ppAuxDecls && localDecl.isAuxDecl || hidden.contains localDecl.fvarId then
5923         pure (varNames, prevType?, fmt)
5924     else
5925         ppVars varNames prevType? fmt localDecl
5926 let fmt ← pushPending varNames type? fmt
5927 let fmt := addLine fmt
5928 let typeFmt ← ppExpr mvarDecl.type
5929 let fmt := fmt ++ "⊢" ++ " " ++ Format.nest indent typeFmt
5930 match mvarDecl.userName with
5931 | Name.anonymous => pure fmt
5932 | name           => return "case " ++ format name.eraseMacroScopes ++ Format.line ++ fmt
5933 end Lean.Meta
5934 // ::::::::::::::
5935 // RecursorInfo.lean
5936 // ::::::::::::::
5937 /-
5938 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
5939 Released under Apache 2.0 license as described in the file LICENSE.
5940 Authors: Leonardo de Moura
5941 -/
5942 import Lean.AuxRecursor
5943 import Lean.Util.FindExpr
5944 import Lean.Meta.ExprDefEq
5945

```

```

5946 namespace Lean.Meta
5947
5948 inductive RecursorUnivLevelPos where
5949   | motive          -- marks where the universe of the motive should go
5950   | majorType (idx : Nat) -- marks where the #idx universe of the major premise type goes
5951
5952 instance : ToString RecursorUnivLevelPos := {fun
5953   | RecursorUnivLevelPos.motive      => "<motive-univ>"
5954   | RecursorUnivLevelPos.majorType idx => toString idx}
5955
5956 structure RecursorInfo where
5957   recursorName  : Name
5958   typeName      : Name
5959   univLevelPos  : List RecursorUnivLevelPos
5960   depElim       : Bool
5961   recursive     : Bool
5962   numArgs       : Nat -- Total number of arguments
5963   majorPos      : Nat
5964   paramsPos     : List (Option Nat) -- Position of the recursor parameters in the major premise, instance implicit arguments are `none`
5965   indicesPos    : List Nat -- Position of the recursor indices in the major premise
5966   produceMotive : List Bool -- If the i-th element is true then i-th minor premise produces the motive
5967
5968 namespace RecursorInfo
5969
5970 def numParams (info : RecursorInfo) : Nat := info.paramsPos.length
5971 def numIndices (info : RecursorInfo) : Nat := info.indicesPos.length
5972 def motivePos (info : RecursorInfo) : Nat := info.numParams
5973 def firstIndexPos (info : RecursorInfo) : Nat := info.majorPos - info.numIndices
5974
5975 def isMinor (info : RecursorInfo) (pos : Nat) : Bool :=
5976   if pos ≤ info.motivePos then false
5977   else if info.firstIndexPos ≤ pos && pos ≤ info.majorPos then false
5978   else true
5979
5980 def numMinors (info : RecursorInfo) : Nat :=
5981   let r := info.numArgs
5982   let r := r - info.motivePos - 1
5983   r - (info.majorPos + 1 - info.firstIndexPos)
5984
5985 instance : ToString RecursorInfo := {fun info =>
5986   "{\n" ++
5987   "  name      := " ++ toString info.recursorName ++ "\n" ++
5988   "  type      := " ++ toString info.typeName ++ "\n" ++
5989   "  univs     := " ++ toString info.univLevelPos ++ "\n" ++
5990   "  depElim   := " ++ toString info.depElim ++ "\n" ++
5991   "  recursive := " ++ toString info.recursive ++ "\n" ++
5992   "  numArgs   := " ++ toString info.numArgs ++ "\n" ++

```



```

5993 " numParams      := " ++ toString info.numParams ++ "\n" ++
5994 " numIndices     := " ++ toString info.numIndices ++ "\n" ++
5995 " numMinors      := " ++ toString info.numMinors ++ "\n" ++
5996 " major          := " ++ toString info.majorPos ++ "\n" ++
5997 " motive         := " ++ toString info.motivePos ++ "\n" ++
5998 " paramsAtMajor  := " ++ toString info.paramsPos ++ "\n" ++
5999 " indicesAtMajor := " ++ toString info.indicesPos ++ "\n" ++
6000 " produceMotive  := " ++ toString info.produceMotive ++ "\n" ++
6001 "}"})
6002
6003 end RecursorInfo
6004
6005 private def mkRecursorInfoForKernelRec (declName : Name) (val : RecursorVal) : MetaM RecursorInfo := do
6006   let ival ← getConstInfoInduct val.getInduct
6007   let numLParams := ival.levelParams.length
6008   let univLevelPos := (List.range numLParams).map RecursorUnivLevelPos.majorType
6009   let univLevelPos := if val.levelParams.length == numLParams then univLevelPos else RecursorUnivLevelPos.motive :: univLevelPos
6010   let produceMotive := List.replicate val.numMinors true
6011   let paramsPos := (List.range val.numParams).map some
6012   let indicesPos := (List.range val.numIndices).map fun pos => val.numParams + pos
6013   let numArgs := val.numIndices + val.numParams + val.numMinors + val.numMotives + 1
6014   pure {
6015     recursorName := declName,
6016     typeName     := val.getInduct,
6017     univLevelPos := univLevelPos,
6018     majorPos     := val.getMajorIdx,
6019     depElim      := true,
6020     recursive    := ival.isRec,
6021     produceMotive := produceMotive,
6022     paramsPos    := paramsPos,
6023     indicesPos   := indicesPos,
6024     numArgs      := numArgs
6025   }
6026
6027
6028 private def getMajorPosIfAuxRecursor? (declName : Name) (majorPos? : Option Nat) : MetaM (Option Nat) :=
6029   if majorPos?.isSome then pure majorPos?
6030   else do
6031     let env ← getEnv
6032     if !isAuxRecursor env declName then pure none
6033     else match declName with
6034     | Name.str p s _ =>
6035       if s != recOnSuffix && s != casesOnSuffix && s != brecOnSuffix then
6036         pure none
6037       else do
6038         let val ← getConstInfoRec (mkRecName p)
6039         pure $ some (val.numParams + val.numIndices + (if s == casesOnSuffix then 1 else val.numMotives))

```

```

6040 | _ => pure none
6041
6042 private def checkMotive (declName : Name) (motive : Expr) (motiveArgs : Array Expr) : MetaM Unit :=
6043   unless motive.isFVar && motiveArgs.all Expr.isFVar do
6044     throwError! "invalid user defined recursor '{declName}', result type must be of the form (C t), where C is a bound variable, and t
is a (possibly empty) sequence of bound variables"
6045
6046 /- Compute number of parameters for (user-defined) recursor.
6047   We assume a parameter is anything that occurs before the motive -/
6048 private partial def getNumParams (xs : Array Expr) (motive : Expr) (i : Nat) : Nat :=
6049   if h : i < xs.size then
6050     let x := xs.get (i, h)
6051     if motive == x then i
6052     else getNumParams xs motive (i+1)
6053   else
6054     i
6055
6056 private def getMajorPosDepElim (declName : Name) (majorPos? : Option Nat) (xs : Array Expr) (motive : Expr) (motiveArgs : Array Expr)
: MetaM (Expr × Nat × Bool) := do
6057   match majorPos? with
6058   | some majorPos =>
6059     if h : majorPos < xs.size then
6060       let major := xs.get (majorPos, h)
6061       let depElim := motiveArgs.contains major
6062       pure (major, majorPos, depElim)
6063     else throwError! "invalid major premise position for user defined recursor, recursor has only {xs.size} arguments"
6064   | none => do
6065     if motiveArgs.isEmpty then
6066       throwError! "invalid user defined recursor, '{declName}' does not support dependent elimination, and position of the major
premise was not specified (solution: set attribute '[recursor <pos>]', where <pos> is the position of the major premise)"
6067     let major := motiveArgs.back
6068     match xs.getIdx? major with
6069     | some majorPos => pure (major, majorPos, true)
6070     | none          => throwError! "ill-formed recursor '{declName}'"
6071
6072 private def getParamsPos (declName : Name) (xs : Array Expr) (numParams : Nat) (Iargs : Array Expr) : MetaM (List (Option Nat)) := do
6073   let mut paramsPos := #[]
6074   for i in [:numParams] do
6075     let x := xs[i]
6076     match (← Iargs.findIdxM? fun Iarg => isDefEq Iarg x) with
6077     | some j => paramsPos := paramsPos.push (some j)
6078     | none   => do
6079       let localDecl ← getLocalDecl x.fvarId!
6080       if localDecl.binderInfo.isInstImplicit then
6081         paramsPos := paramsPos.push none
6082       else
6083         throwError! "invalid user defined recursor '{declName}', type of the major premise does not contain the recursor parameter"

```

```

6085 pure paramsPos.toList
6086
6087 private def getIndicesPos (declName : Name) (xs : Array Expr) (majorPos numIndices : Nat) (Iargs : Array Expr) : MetaM (List Nat) := do
6088   let mut indicesPos := #[]
6089   for i in [:numIndices] do
6090     let i := majorPos - numIndices + i
6091     let x := xs[i]
6092     match (← Iargs.findIdxM? fun Iarg => isDefEq Iarg x) with
6093     | some j => indicesPos := indicesPos.push j
6094     | none   => throwError! "invalid user defined recursor '{declName}', type of the major premise does not contain the recursor index"
6095   pure indicesPos.toList
6096
6097 private def getMotiveLevel (declName : Name) (motiveResultType : Expr) : MetaM Level :=
6098   match motiveResultType with
6099   | Expr.sort u@(Level.zero _) _ => pure u
6100   | Expr.sort u@(Level.param _ _) _ => pure u
6101   | _ =>
6102     throwError! "invalid user defined recursor '{declName}', motive result sort must be Prop or (Sort u) where u is a universe level
parameter"
6103
6104 private def getUnivLevelPos (declName : Name) (lparams : List Name) (motiveLvl : Level) (Ilevels : List Level) : MetaM (List
RecursorUnivLevelPos) := do
6105   let Ilevels := Ilevels.toArray
6106   let mut univLevelPos := #[]
6107   for p in lparams do
6108     if motiveLvl == mkLevelParam p then
6109       univLevelPos := univLevelPos.push RecursorUnivLevelPos.motive
6110     else
6111       match Ilevels.findIdx? fun u => u == mkLevelParam p with
6112       | some i => univLevelPos := univLevelPos.push (RecursorUnivLevelPos.majorType i)
6113       | none   =>
6114         throwError! "invalid user defined recursor '{declName}', major premise type does not contain universe level parameter '{p}'"
6115   pure univLevelPos.toList
6116
6117 private def getProduceMotiveAndRecursive (xs : Array Expr) (numParams numIndices majorPos : Nat) (motive : Expr) : MetaM (List Bool ×
Bool) := do
6118   let mut produceMotive := #[]
6119   let mut recursor      := false
6120   for i in [:xs.size] do
6121     if i < numParams + 1 then
6122       continue --skip parameters and motive
6123     if majorPos - numIndices ≤ i && i ≤ majorPos then
6124       continue -- skip indices and major premise
6125     -- process minor premise
6126     let x := xs[i]
6127     let xType ← inferType x
6128     (produceMotive, recursor) ← forallTelescopeReducing xType fun minorArgs minorResultType => minorResultType.withApp fun res _ => do

```

```

6129     let produceMotive := produceMotive.push (res == motive)
6130     let recursor ← if recursor then pure recursor else minorArgs.anyM fun minorArg => do
6131       let minorArgType ← inferType minorArg
6132       pure (minorArgType.find? fun e => e == motive).isSome
6133     pure (produceMotive, recursor)
6134   pure (produceMotive.toList, recursor)
6135
6136 private def checkMotiveResultType (declName : Name) (motiveArgs : Array Expr) (motiveResultType : Expr) (motiveTypeParams : Array Expr)
: MetaM Unit := do
6137   if !motiveResultType.isSort || motiveArgs.size != motiveTypeParams.size then
6138     throwError! "invalid user defined recursor '{declName}', motive must have a type of the form (C : Pi (i : B A), I A i -> Type),
where A is (possibly empty) sequence of variables (aka parameters), (i : B A) is a (possibly empty) telescope (aka indices), and I is a
constant"
6139
6140 private def mkRecursorInfoAux (cinfo : ConstantInfo) (majorPos? : Option Nat) : MetaM RecursorInfo := do
6141   let declName := cinfo.name
6142   let majorPos? ← getMajorPosIfAuxRecursor? declName majorPos?
6143   forallTelescopeReducing cinfo.type fun xs type => type.withApp fun motive motiveArgs => do
6144     checkMotive declName motive motiveArgs
6145     let numParams := getNumParams xs motive 0
6146     let (major, majorPos, depElim) ← getMajorPosDepElim declName majorPos? xs motive motiveArgs
6147     let numIndices := if depElim then motiveArgs.size - 1 else motiveArgs.size
6148     if majorPos < numIndices then
6149       throwError! "invalid user defined recursor '{declName}', indices must occur before major premise"
6150     let majorType ← inferType major
6151     majorType.withApp fun I Iargs =>
6152       match I with
6153       | Expr.const Iname Ilevels _ => do
6154         let paramsPos ← getParamsPos declName xs numParams Iargs
6155         let indicesPos ← getIndicesPos declName xs majorPos numIndices Iargs
6156         let motiveType ← inferType motive
6157         forallTelescopeReducing motiveType fun motiveTypeParams motiveResultType => do
6158           checkMotiveResultType declName motiveArgs motiveResultType motiveTypeParams
6159           let motiveLvl ← getMotiveLevel declName motiveResultType
6160           let univLevelPos ← getUnivLevelPos declName cinfo.levelParams motiveLvl Ilevels
6161           let (produceMotive, recursive) ← getProduceMotiveAndRecursive xs numParams numIndices majorPos motive
6162           pure {
6163             recursorName := declName,
6164             typeName      := Iname,
6165             univLevelPos  := univLevelPos,
6166             majorPos      := majorPos,
6167             depElim       := depElim,
6168             recursive     := recursive,
6169             produceMotive := produceMotive,
6170             paramsPos     := paramsPos,
6171             indicesPos    := indicesPos,
6172             numArgs       := xs.size

```

```

6173     }
6174     | _ => throwError! "invalid user defined recursor '{declName}', type of the major premise must be of the form (I ...), where I is a
constant"
6175
6176 /-
6177 @[builtinAttrParser] def «recursor» := parser! "recursor " >> numLit
6178 -/
6179 def Attribute.Recursor.getMajorPos (stx : Syntax) : AttrM Nat := do
6180   if stx.getKind == `Lean.Parser.Attr.recursor then
6181     let pos := stx[1].isNatLit?.getD 0
6182     if pos == 0 then
6183       throwErrorAt! stx "major premise position must be greater than zero"
6184     return pos - 1
6185   else
6186     throwErrorAt! stx "unexpected attribute argument, numeral expected"
6187
6188 private def mkRecursorInfoCore (declName : Name) (majorPos? : Option Nat := none) : MetaM RecursorInfo := do
6189   let cinfo ← getConstInfo declName
6190   match cinfo with
6191   | ConstantInfo.recInfo val => mkRecursorInfoForKernelRec declName val
6192   | _                        => mkRecursorInfoAux cinfo majorPos?
6193
6194 builtin_initialize recursorAttribute : ParametricAttribute Nat ←
6195   registerParametricAttribute {
6196     name := `recursor,
6197     descr := "user defined recursor, numerical parameter specifies position of the major premise",
6198     getParam := fun _ stx => Attribute.Recursor.getMajorPos stx
6199     afterSet := fun declName majorPos => do
6200       discard <| mkRecursorInfoCore declName (some majorPos) |>.run'
6201   }
6202
6203 def getMajorPos? (env : Environment) (declName : Name) : Option Nat :=
6204   recursorAttribute.getParam env declName
6205
6206 def mkRecursorInfo (declName : Name) (majorPos? : Option Nat := none) : MetaM RecursorInfo := do
6207   let cinfo ← getConstInfo declName
6208   match cinfo with
6209   | ConstantInfo.recInfo val => mkRecursorInfoForKernelRec declName val
6210   | _                        => match majorPos? with
6211     | none => do mkRecursorInfoAux cinfo (getMajorPos? (← getEnv) declName)
6212     | _    => mkRecursorInfoAux cinfo majorPos?
6213
6214 end Lean.Meta
6215 // ::::::::::::::
6216 // ReduceEval.lean
6217 // ::::::::::::::
6218 /-

```

```

6219 Copyright (c) 2020 Sebastian Ullrich. All rights reserved.
6220 Released under Apache 2.0 license as described in the file LICENSE.
6221 Authors: Sebastian Ullrich
6222 -/
6223
6224 /-! Evaluation by reduction -/
6225
6226 import Lean.Meta.Offset
6227
6228 namespace Lean.Meta
6229
6230 class ReduceEval (α : Type) where
6231   reduceEval : Expr → MetaM α
6232
6233 def reduceEval [ReduceEval α] (e : Expr) : MetaM α :=
6234   withAtLeastTransparency TransparencyMode.default $
6235     ReduceEval.reduceEval e
6236
6237 private def throwFailedToEval (e : Expr) : MetaM α :=
6238   throwError! "reduceEval: failed to evaluate argument{indentExpr e}"
6239
6240 instance : ReduceEval Nat where
6241   reduceEval e := do
6242     let e ← whnf e
6243     let some n ← evalNat e | throwFailedToEval e
6244     pure n
6245
6246 instance [ReduceEval α] : ReduceEval (Option α) where
6247   reduceEval e := do
6248     let e ← whnf e
6249     let Expr.const c .. ← pure e.getAppFn | throwFailedToEval e
6250     let nargs := e.getAppNumArgs
6251     if c == `Option.none && nargs == 0 then pure none
6252     else if c == `Option.some && nargs == 1 then some <$> reduceEval e.appArg!
6253     else throwFailedToEval e
6254
6255 instance : ReduceEval String where
6256   reduceEval e := do
6257     let Expr.lit (Literal.strVal s) _ ← whnf e | throwFailedToEval e
6258     pure s
6259
6260 private partial def evalName (e : Expr) : MetaM Name := do
6261   let e ← whnf e
6262   let Expr.const c _ _ ← pure e.getAppFn | throwFailedToEval e
6263   let nargs := e.getAppNumArgs
6264   if c == `Lean.Name.anonymous && nargs == 0 then pure Name.anonymous
6265   else if c == `Lean.Name.str && nargs == 3 then do

```

```

6266     let n ← evalName $ e.getArg! 0
6267     let s ← reduceEval $ e.getArg! 1
6268     pure $ Name.mkStr n s
6269   else if c == `Lean.Name.num && nargs == 3 then do
6270     let n ← evalName $ e.getArg! 0
6271     let u ← reduceEval $ e.getArg! 1
6272     pure $ Name.mkNum n u
6273   else
6274     throwFailedToEval e
6275
6276 instance : ReduceEval Name where
6277   reduceEval := evalName
6278
6279 end Lean.Meta
6280 // ::::::::::::::
6281 // Reduce.lean
6282 // ::::::::::::::
6283 /-
6284 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6285 Released under Apache 2.0 license as described in the file LICENSE.
6286 Authors: Leonardo de Moura
6287 -/
6288 import Lean.Meta.Basic
6289 import Lean.Meta.FunInfo
6290 import Lean.Util.MonadCache
6291
6292 namespace Lean.Meta
6293
6294 partial def reduce (e : Expr) (explicitOnly skipTypes skipProofs := true) : MetaM Expr :=
6295   let rec visit (e : Expr) : MonadCacheT Expr Expr MetaM Expr :=
6296     checkCache e fun _ => Core.withIncRecDepth do
6297       if (← (skipTypes <&& isType e)) then
6298         return e
6299       else if (← (skipProofs <&& isProof e)) then
6300         return e
6301       else
6302         let e ← whnf e
6303         match e with
6304         | Expr.app .. =>
6305           let f      := e.getAppFn
6306           let nargs  := e.getAppNumArgs
6307           let finfo ← getFunInfoNArgs f nargs
6308           let mut args := e.getAppArgs
6309           for i in [:args.size] do
6310             if i < finfo.paramInfo.size then
6311               let info := finfo.paramInfo[i]
6312               if !explicitOnly || info.isExplicit then

```

```

6313         args ← args.modifyM i visit
6314     else
6315         args ← args.modifyM i visit
6316     pure (mkAppN f args)
6317 | Expr.lam ..      => lambdaTelescope e fun xs b => do mkLambdaFVars xs (← visit b)
6318 | Expr.forallE ..  => forallTelescope e fun xs b => do mkForallFVars xs (← visit b)
6319 |                 => return e
6320 visit e |>.run
6321
6322 end Lean.Meta
6323 // ::::::::::::::
6324 // SizeOf.lean
6325 // ::::::::::::::
6326 /-
6327 Copyright (c) 2021 Microsoft Corporation. All rights reserved.
6328 Released under Apache 2.0 license as described in the file LICENSE.
6329 Authors: Leonardo de Moura
6330 -/
6331 import Lean.Meta.AppBuilder
6332 import Lean.Meta.Instances
6333
6334 namespace Lean.Meta
6335
6336 /-- Create `SizeOf` local instances for applicable parameters, and execute `k` using them. -/
6337 private partial def mkLocalInstances {α} (params : Array Expr) (k : Array Expr → MetaM α) : MetaM α :=
6338   loop 0 #[]
6339 where
6340   loop (i : Nat) (insts : Array Expr) : MetaM α := do
6341     if i < params.size then
6342       let param := params[i]
6343       let paramType ← inferType param
6344       let instType? ← forallTelescopeReducing paramType fun xs _ => do
6345         let type ← mkAppN param xs
6346         try
6347           let sizeOf ← mkAppM `SizeOf #[type]
6348           let instType ← mkForallFVars xs sizeOf
6349           return some instType
6350       catch _ =>
6351         return none
6352     match instType? with
6353     | none => loop (i+1) insts
6354     | some instType =>
6355       let instName ← mkFreshUserName `inst
6356       withLocalDecl instName BinderInfo.instImplicit instType fun inst =>
6357         loop (i+1) (insts.push inst)
6358   else
6359     k insts

```



```

6360
6361 /--
6362   Return `some x` if `fvar` has type of the form `... -> motive ... fvar` where `motive` in `motiveFVars`.
6363   That is, `x` "produces" one of the recursor motives.
6364 -/
6365 private def isInductiveHypothesis? (motiveFVars : Array Expr) (fvar : Expr) : MetaM (Option Expr) := do
6366   forallTelescopeReducing (← inferType fvar) fun _ type =>
6367     if type.isApp && motiveFVars.contains type.getAppFn then
6368       return some type.appArg!
6369     else
6370       return none
6371
6372 private def isInductiveHypothesis (motiveFVars : Array Expr) (fvar : Expr) : MetaM Bool :=
6373   return (← isInductiveHypothesis? motiveFVars fvar).isSome
6374
6375 /--
6376   Let `motiveFVars` be free variables for each motive in a kernel recursor, and `minorFVars` the free variables for a minor premise.
6377   Then, return `some idx` if `minorFVars[idx]` has a type of the form `... -> motive ... fvar` for some `motive` in `motiveFVars`.
6378 -/
6379 private def isRecField? (motiveFVars : Array Expr) (minorFVars : Array Expr) (fvar : Expr) : MetaM (Option Nat) := do
6380   let mut idx := 0
6381   for minorFVar in minorFVars do
6382     if let some fvar' ← isInductiveHypothesis? motiveFVars minorFVar then
6383       if fvar == fvar' then
6384         return some idx
6385     idx := idx + 1
6386   return none
6387
6388 private partial def mkSizeOfMotives {α} (motiveFVars : Array Expr) (k : Array Expr → MetaM α) : MetaM α :=
6389   loop 0 #[]
6390 where
6391   loop (i : Nat) (motives : Array Expr) : MetaM α := do
6392     if i < motiveFVars.size then
6393       let type ← inferType motiveFVars[i]
6394       let motive ← forallTelescopeReducing type fun xs _ => do
6395         mkLambdaFVars xs <| mkConst ``Nat
6396         trace[Meta.sizeOf]! "motive: {motive}"
6397         loop (i+1) (motives.push motive)
6398     else
6399       k motives
6400
6401 private partial def mkSizeOfMinors {α} (motiveFVars : Array Expr) (minorFVars : Array Expr) (minorFVars' : Array Expr) (k : Array Expr
→ MetaM α) : MetaM α :=
6402   assert! minorFVars.size == minorFVars'.size
6403   loop 0 #[]
6404 where
6405   loop (i : Nat) (minors : Array Expr) : MetaM α := do

```

```

6406   if i < minorFVars.size then
6407     forallTelescopeReducing (← inferType minorFVars[i]) fun xs _ =>
6408     forallBoundedTelescope (← inferType minorFVars'[i]) xs.size fun xs' _ => do
6409       let mut minor ← mkNumeral (mkConst ``Nat) 1
6410       for x in xs, x' in xs' do
6411         unless (← isInductiveHypothesis motiveFVars x) do
6412           unless (← whnf (← inferType x)).isForall do -- we suppress higher-order fields
6413             match (← isRecField? motiveFVars xs x) with
6414             | some idx => minor ← mkAdd minor xs'[idx]
6415             | none     => minor ← mkAdd minor (← mkAppM ``SizeOf.sizeOf #[x'])
6416       minor ← mkLambdaFVars xs' minor
6417       trace[Meta.sizeOf]! "minor: {minor}"
6418       loop (i+1) (minors.push minor)
6419   else
6420     k minors
6421
6422 /--
6423   Create a "sizeOf" function with name `declName` using the recursor `recName`.
6424 -/
6425 partial def mkSizeOfFn (recName : Name) (declName : Name): MetaM Unit := do
6426   trace[Meta.sizeOf]! "recName: {recName}"
6427   let recInfo : RecursorVal ← getConstInfoRec recName
6428   forallTelescopeReducing recInfo.type fun xs type =>
6429     let levelParams := recInfo.levelParams.tail! -- universe parameters for declaration being defined
6430     let params := xs[:recInfo.numParams]
6431     let motiveFVars := xs[recInfo.numParams : recInfo.numParams + recInfo.numMotives]
6432     let minorFVars := xs[recInfo.getFirstMinorIdx : recInfo.getFirstMinorIdx + recInfo.numMinors]
6433     let indices := xs[recInfo.getFirstIndexIdx : recInfo.getFirstIndexIdx + recInfo.numIndices]
6434     let major := xs[recInfo.getMajorIdx]
6435     let nat := mkConst ``Nat
6436     mkLocalInstances params fun localInsts =>
6437     mkSizeOfMotives motiveFVars fun motives => do
6438       let us := levelOne :: levelParams.map mkLevelParam -- universe level parameters for `rec`-application
6439       let recFn := mkConst recName us
6440       let val := mkAppN recFn (params ++ motives)
6441       forallBoundedTelescope (← inferType val) recInfo.numMinors fun minorFVars' _ =>
6442       mkSizeOfMinors motiveFVars minorFVars minorFVars' fun minors => do
6443         let sizeOfParams := params ++ localInsts ++ indices ++ #[major]
6444         let sizeOfType ← mkForallFVars sizeOfParams nat
6445         let val := mkAppN val (minors ++ indices ++ #[major])
6446         trace[Meta.sizeOf]! "val: {val}"
6447         let sizeOfValue ← mkLambdaFVars sizeOfParams val
6448         addDecl <| Declaration.defnDecl {
6449           name      := declName
6450           levelParams := levelParams
6451           type      := sizeOfType
6452           value     := sizeOfValue

```

```

6453     safety      := DefinitionSafety.safe
6454     hints        := ReducibilityHints.abbrev
6455   }
6456
6457 /--
6458   Create `sizeof` functions for all inductive datatypes in the mutual inductive declaration containing `typeName`
6459   The resulting array contains the generated functions names. The `NameMap` maps recursor names into the generated function names.
6460   There is a function for each element of the mutual inductive declaration, and for auxiliary recursors for nested inductive types.
6461 -/
6462 def mkSizeOfFns (typeName : Name) : MetaM (Array Name × NameMap Name) := do
6463   let indInfo ← getConstInfoInduct typeName
6464   let recInfo ← getConstInfoRec (mkRecName typeName)
6465   let numExtra := recInfo.numMotives - indInfo.all.length -- numExtra > 0 for nested inductive types
6466   let mut result := #[]
6467   let baseName := indInfo.all.head! ++ `_sizeof` -- we use the first inductive type as the base name for `sizeof` functions
6468   let mut i := 1
6469   let mut recMap : NameMap Name := {}
6470   for indTypeName in indInfo.all do
6471     let sizeofName := baseName.appendIndexAfter i
6472     let recName := mkRecName indTypeName
6473     mkSizeOfFn recName sizeofName
6474     recMap := recMap.insert recName sizeofName
6475     result := result.push sizeofName
6476     i := i + 1
6477   for j in [:numExtra] do
6478     let recName := (mkRecName indInfo.all.head!).appendIndexAfter (j+1)
6479     let sizeofName := baseName.appendIndexAfter i
6480     mkSizeOfFn recName sizeofName
6481     recMap := recMap.insert recName sizeofName
6482     result := result.push sizeofName
6483     i := i + 1
6484   return (result, recMap)
6485
6486 def mkSizeOfSpecLemmaName (ctorName : Name) : Name :=
6487   ctorName ++ `_sizeof_spec`
6488
6489 def mkSizeOfSpecLemmaInstance (ctorApp : Expr) : MetaM Expr :=
6490   matchConstCtor ctorApp.getAppFn (fun _ => throwError! "failed to apply 'sizeof' spec, constructor expected{indentExpr ctorApp}") fun
6491   ctorInfo ctorLevels => do
6492     let ctorArgs      := ctorApp.getAppArgs
6493     let ctorFields    := ctorArgs[ctorArgs.size - ctorInfo.numFields:]
6494     let lemmaName     := mkSizeOfSpecLemmaName ctorInfo.name
6495     let lemmaInfo     ← getConstInfo lemmaName
6496     let lemmaArity    ← forallTelescopeReducing lemmaInfo.type fun xs _ => return xs.size
6497     let lemmaArgMask := mkArray (lemmaArity - ctorInfo.numFields) (none (α := Expr))
6498     let lemmaArgMask := lemmaArgMask ++ ctorFields.toArray.map some
6499     mkAppOptM lemmaName lemmaArgMask

```

```

6499
6500 /- SizeOf spec theorem for nested inductive types -/
6501 namespace SizeOfSpecNested
6502
6503 structure Context where
6504   indInfo      : InductiveVal
6505   sizeOfFns    : Array Name
6506   ctorName     : Name
6507   params       : Array Expr
6508   localInsts   : Array Expr
6509   recMap       : NameMap Name -- mapping from recursor name into `_sizeOf_<idx>` function name (see `mkSizeOfFns`)
6510
6511 abbrev M := ReaderT Context MetaM
6512
6513 def throwUnexpected {α} (msg : MessageData) : M α := do
6514   throwError! "failed to generate sizeOf lemma for {(← read).ctorName} (use `set_option genSizeOfSpec false` to disable lemma
6515   generation), {msg}"
6516
6517 def throwFailed {α} : M α := do
6518   throwError! "failed to generate sizeOf lemma for {(← read).ctorName}, (use `set_option genSizeOfSpec false` to disable lemma
6519   generation)"
6520
6521 /- Convert a recursor application into a `_sizeOf_<idx>` application. -/
6522 private def recToSizeOf (e : Expr) : M Expr := do
6523   matchConstRec e.getAppFn (fun _ => throwFailed) fun info us => do
6524     match (← read).recMap.find? info.name with
6525     | none => throwUnexpected m!"expected recursor application {indentExpr e}"
6526     | some sizeOfName =>
6527       let args      := e.getAppArgs
6528       let indices   := args[info.getFirstIndexIdx : info.getFirstIndexIdx + info.numIndices]
6529       let major     := args[info.getMajorIdx]
6530       return mkAppN (mkConst sizeOfName us.tail!) ((← read).params ++ (← read).localInsts ++ indices ++ #[major])
6531
6532 mutual
6533 /- Construct minor premise proof for `mkSizeOfAuxLemmaProof`. `ys` contains fields and inductive hypotheses for the minor premise.
6534 -/
6535 private partial def mkMinorProof (ys : Array Expr) (lhs rhs : Expr) : M Expr := do
6536   trace[Meta.sizeOf.minor]! "{lhs} =?= {rhs}"
6537   if (← isDefEq lhs rhs) then
6538     mkEqRefl rhs
6539   else
6540     match (← whnfI lhs).natAdd?, (← whnfI rhs).natAdd? with
6541     | some (a₁, b₁), some (a₂, b₂) =>
6542       let p₁ ← mkMinorProof ys a₁ a₂
6543       let p₂ ← mkMinorProofStep ys b₁ b₂
6544       mkCongr (← mkCongrArg (mkConst ``Nat.add) p₁) p₂
6545     | _, _ =>

```

```

6543         throwUnexpected m!"expected 'Nat.add' application, lhs is {indentExpr lhs}\nrhs is{indentExpr rhs}"
6544
6545 /--
6546   Helper method for `mkMinorProof`. The proof step is one of the following
6547   - Reflexivity
6548   - Assumption (i.e., using an inductive hypotheses from `ys`)
6549   - `mkSizeOfAuxLemma` application. This case happens when we have multiple levels of nesting
6550 -/
6551 private partial def mkMinorProofStep (ys : Array Expr) (lhs rhs : Expr) : M Expr := do
6552   if (← isDefEq lhs rhs) then
6553     mkEqRefl rhs
6554   else
6555     let lhs ← recToSizeOf lhs
6556     trace[Meta.sizeOf.minor.step]! "{lhs} =?= {rhs}"
6557     let target ← mkEq lhs rhs
6558     for y in ys do
6559       if (← isDefEq (← inferType y) target) then
6560         return y
6561     mkSizeOfAuxLemma lhs rhs
6562
6563 /-- Construct proof of auxiliary lemma. See `mkSizeOfAuxLemma` -/
6564 private partial def mkSizeOfAuxLemmaProof (info : InductiveVal) (lhs rhs : Expr) : M Expr := do
6565   let lhsArgs := lhs.getAppArgs
6566   let sizeOfBaseArgs := lhsArgs[:lhsArgs.size - info.numIndices - 1]
6567   let indicesMajor := lhsArgs[lhsArgs.size - info.numIndices - 1:]
6568   let sizeOfLevels := lhs.getAppFn.constLevels!
6569   /- Auxiliary function for constructing an `_sizeOf_{idx}` for `ys`,
6570      where `ys` are the indices + major.
6571      Recall that if `info.name` is part of a mutually inductive declaration, then the resulting application
6572      is not necessarily a `lhs.getAppFn` application.
6573      The result is an application of one of the `(← read).sizeOfFns` functions.
6574      We use this auxiliary function to builtin the motive of the recursor. -/
6575   let rec mkSizeOf (ys : Array Expr) : M Expr := do
6576     for sizeOfFn in (← read).sizeOfFns do
6577       let candidate := mkAppN (mkAppN (mkConst sizeOfFn sizeOfLevels) sizeOfBaseArgs) ys
6578       if (← isTypeCorrect candidate) then
6579         return candidate
6580     throwFailed
6581   let major := lhs.appArg!
6582   let majorType ← whnf (← inferType major)
6583   let majorTypeArgs := majorType.getAppArgs
6584   match majorType.getAppFn.const? with
6585   | none => throwFailed
6586   | some (_, us) =>
6587     let recName := mkRecName info.name
6588     let recInfo ← getConstInfoRec recName
6589     let r := mkConst recName (levelZero :: us)

```

```

6590 let r := mkAppN r majorTypeArgs[:info.numParams]
6591 forallBoundedTelescope (← inferType r) recInfo.numMotives fun motiveFVars _ => do
6592   let mut r := r
6593   -- Add motives
6594   for motiveFVar in motiveFVars do
6595     let motive ← forallTelescopeReducing (← inferType motiveFVar) fun ys _ => do
6596       let lhs ← mkSizeOf ys
6597       let rhs ← mkAppM ``SizeOf.sizeOf #[ys.back]
6598       mkLambdaFVars ys (← mkEq lhs rhs)
6599     r := mkApp r motive
6600 forallBoundedTelescope (← inferType r) recInfo.numMinors fun minorFVars _ => do
6601   let mut r := r
6602   -- Add minors
6603   for minorFVar in minorFVars do
6604     let minor ← forallTelescopeReducing (← inferType minorFVar) fun ys target => do
6605       let target ← whnf target
6606       match target.eq? with
6607       | none => throwFailed
6608       | some (_, lhs, rhs) =>
6609         if (← isDefEq lhs rhs) then
6610           mkLambdaFVars ys (← mkEqRefl rhs)
6611         else
6612           let lhs ← unfoldDefinition lhs -- Unfold `C._sizeOf_<idx>`
6613           -- rhs is of the form `sizeOf (ctor ...)`
6614           let ctorApp := rhs.appArg!
6615           let specLemma ← mkSizeOfSpecLemmaInstance ctorApp
6616           let specEq ← whnf (← inferType specLemma)
6617           match specEq.eq? with
6618           | none => throwFailed
6619           | some (_, rhs, rhsExpanded) =>
6620             let lhs_eq_rhsExpanded ← mkMinorProof ys lhs rhsExpanded
6621             let rhsExpanded_eq_rhs ← mkEqSymm specLemma
6622             mkLambdaFVars ys (← mkEqTrans lhs_eq_rhsExpanded rhsExpanded_eq_rhs)
6623     r := mkApp r minor
6624   -- Add indices and major
6625   return mkAppN r indicesMajor
6626
6627 /--
6628 Generate proof for `C._sizeOf_<idx> t = sizeOf t` where `C._sizeOf_<idx>` is a auxiliary function
6629 generated for a nested inductive type in `C`.
6630 For example, given
6631 ```lean
6632 inductive Expr where
6633   | app (f : String) (args : List Expr)
6634   ...
6635 We generate the auxiliary function `Expr._sizeOf_1 : List Expr → Nat`.
6636 To generate the `sizeOf` spec lemma

```

```

6637   ```
6638   sizeOf (Expr.app f args) = 1 + sizeOf f + sizeOf args
6639   ```
6640   we need an auxiliary lemma for showing `Expr._sizeOf_1 args = sizeOf args`.
6641   Recall that `sizeOf (Expr.app f args)` is definitionally equal to `1 + sizeOf f + Expr._sizeOf_1 args`, but
6642   `Expr._sizeOf_1 args` is not definitionally equal to `sizeOf args`. We need a proof by induction.
6643   -/
6644   private partial def mkSizeOfAuxLemma (lhs rhs : Expr) : M Expr := do
6645     trace[Meta.sizeOf.aux]! "{lhs} =?= {rhs}"
6646     match lhs.getAppFn.const? with
6647     | none => throwFailed
6648     | some (fName, us) =>
6649       let thmLevelParams ← us.mapM fun
6650         | Level.param n _ => return n
6651         | _ => throwFailed
6652       let thmName := fName.appendAfter "_eq"
6653       if (← getEnv).contains thmName then
6654         -- Auxiliary lemma has already been defined
6655         return mkAppN (mkConst thmName us) lhs.getAppArgs
6656       else
6657         -- Define auxiliary lemma
6658         -- First, generalize indices
6659         let x := lhs.appArg!
6660         let xType ← whnf (← inferType x)
6661         matchConstInduct xType.getAppFn (fun _ => throwFailed) fun info _ => do
6662           let params := xType.getAppArgs[:info.numParams]
6663           forallTelescopeReducing (← inferType (mkAppN xType.getAppFn params)) fun indices _ => do
6664             let majorType := mkAppN (mkAppN xType.getAppFn params) indices
6665             withLocalDeclD `x majorType fun major => do
6666               let lhsArgs := lhs.getAppArgs
6667               let lhsArgsNew := lhsArgs[:lhsArgs.size - 1 - indices.size] ++ indices ++ #[major]
6668               let lhsNew := mkAppN lhs.getAppFn lhsArgsNew
6669               let rhsNew ← mkAppM ``SizeOf.sizeOf #[major]
6670               let eq ← mkEq lhsNew rhsNew
6671               let thmParams := lhsArgsNew
6672               let thmType ← mkForallFVars thmParams eq
6673               let thmValue ← mkSizeOfAuxLemmaProof info lhsNew rhsNew
6674               let thmValue ← mkLambdaFVars thmParams thmValue
6675               trace[Meta.sizeOf]! "thmValue: {thmValue}"
6676               addDecl <| Declaration.thmDecl {
6677                 name      := thmName
6678                 levelParams := thmLevelParams
6679                 type       := thmType
6680                 value      := thmValue
6681               }
6682             return mkAppN (mkConst thmName us) lhs.getAppArgs
6683

```

```

6684 end
6685
6686 /- Prove SizeOf spec lemma of the form `sizeOf <ctor-application> = 1 + sizeOf <field_1> + ... + sizeOf <field_n> -/
6687 partial def main (lhs rhs : Expr) : M Expr := do
6688   if (← isDefEq lhs rhs) then
6689     mkEqRefl rhs
6690   else
6691     /- Expand lhs and rhs to obtain `Nat.add` applications -/
6692     let lhs ← whnfI lhs -- Expand `sizeOf (ctor ...)` into `_sizeOf_<idx>` application
6693     let lhs ← unfoldDefinition lhs -- Unfold `_sizeOf_<idx>` application into `HAdd.hAdd` application
6694     loop lhs rhs
6695 where
6696   loop (lhs rhs : Expr) : M Expr := do
6697     trace[Meta.sizeOf.loop]! "{lhs} =?= {rhs}"
6698     if (← isDefEq lhs rhs) then
6699       mkEqRefl rhs
6700     else
6701       match (← whnfI lhs).natAdd?, (← whnfI rhs).natAdd? with
6702       | some (a₁, b₁), some (a₂, b₂) =>
6703         let p₁ ← loop a₁ a₂
6704         let p₂ ← step b₁ b₂
6705         mkCongr (← mkCongrArg (mkConst ``Nat.add) p₁) p₂
6706       | _, _ =>
6707         throwUnexpected m!"expected 'Nat.add' application, lhs is {indentExpr lhs}\nrhs is {indentExpr rhs}"
6708
6709   step (lhs rhs : Expr) : M Expr := do
6710     if (← isDefEq lhs rhs) then
6711       mkEqRefl rhs
6712     else
6713       let lhs ← recToSizeOf lhs
6714       mkSizeOfAuxLemma lhs rhs
6715
6716 end SizeOfSpecNested
6717
6718 private def mkSizeOfSpecTheorem (indInfo : InductiveVal) (sizeOfFns : Array Name) (recMap : NameMap Name) (ctorName : Name) : MetaM
Unit := do
6719   let ctorInfo ← getConstInfoCtor ctorName
6720   let us := ctorInfo.levelParams.map mkLevelParam
6721   forallTelescopeReducing ctorInfo.type fun xs _ => do
6722     let params := xs[:ctorInfo.numParams]
6723     let fields := xs[ctorInfo.numParams:]
6724     let ctorApp := mkAppN (mkConst ctorName us) xs
6725     mkLocalInstances params fun localInsts => do
6726       let lhs ← mkAppM ``SizeOf.sizeOf #[ctorApp]
6727       let mut rhs ← mkNumeral (mkConst ``Nat) 1
6728       for field in fields do
6729         unless (← whnf (← inferType field)).isForall do

```



```

6730     rhs ← mkAdd rhs (← mkAppM ``SizeOf.sizeOf #[field])
6731   let target ← mkEq lhs rhs
6732   let thmName := mkSizeOfSpecLemmaName ctorName
6733   let thmParams := params ++ localInsts ++ fields
6734   let thmType ← mkForallFVars thmParams target
6735   let thmValue ←
6736     if indInfo.isNested then
6737       SizeOfSpecNested.main lhs rhs |>.run {
6738         indInfo := indInfo, sizeOfFns := sizeOfFns, ctorName := ctorName, params := params, localInsts := localInsts, recMap :=
recMap
6739       }
6740     else
6741       mkEqRefl rhs
6742   let thmValue ← mkLambdaFVars thmParams thmValue
6743   addDecl <| Declaration.thmDecl {
6744     name      := thmName
6745     levelParams := ctorInfo.levelParams
6746     type      := thmType
6747     value     := thmValue
6748   }
6749
6750 private def mkSizeOfSpecTheorems (indTypeNames : Array Name) (sizeOfFns : Array Name) (recMap : NameMap Name) : MetaM Unit := do
6751   for indTypeName in indTypeNames do
6752     let indInfo ← getConstInfoInduct indTypeName
6753     for ctorName in indInfo.ctors do
6754       mkSizeOfSpecTheorem indInfo sizeOfFns recMap ctorName
6755   return ()
6756
6757 register_builtin_option genSizeOf : Bool := {
6758   defValue := true
6759   descr   := "generate `SizeOf` instance for inductive types and structures"
6760 }
6761
6762 register_builtin_option genSizeOfSpec : Bool := {
6763   defValue := true
6764   descr   := "generate `SizeOf` specification theorems for automatically generated instances"
6765 }
6766
6767 def mkSizeOfInstances (typeName : Name) : MetaM Unit := do
6768   if (← getEnv).contains ``SizeOf && genSizeOf.get (← getOptions) && !(← isInductivePredicate typeName) then
6769     let indInfo ← getConstInfoInduct typeName
6770     unless indInfo.isUnsafe do
6771       let (fns, recMap) ← mkSizeOfFns typeName
6772       for indTypeName in indInfo.all, fn in fns do
6773         let indInfo ← getConstInfoInduct indTypeName
6774         forallTelescopeReducing indInfo.type fun xs _ =>
6775           let params := xs[:indInfo.numParams]

```

```

6776     let indices := xs[indInfo.numParams:]
6777     mkLocalInstances params fun localInsts => do
6778       let us := indInfo.levelParams.map mkLevelParam
6779       let indType := mkAppN (mkConst indTypeName us) xs
6780       let sizeOfIndType ← mkAppM ``SizeOf #[indType]
6781       withLocalDeclD `m indType fun m => do
6782         let v ← mkLambdaFVars #[m] <| mkAppN (mkConst fn us) (params ++ localInsts ++ indices ++ #[m])
6783         let sizeOfMk ← mkAppM ``SizeOf.mk #[v]
6784         let instDeclName := indTypeName ++ `_sizeOf_inst
6785         let instDeclType ← mkForallFVars (xs ++ localInsts) sizeOfIndType
6786         let instDeclValue ← mkLambdaFVars (xs ++ localInsts) sizeOfMk
6787         addDecl <| Declaration.defnDecl {
6788           name      := instDeclName
6789           levelParams := indInfo.levelParams
6790           type      := instDeclType
6791           value      := instDeclValue
6792           safety     := DefinitionSafety.safe
6793           hints      := ReducibilityHints.abbrev
6794         }
6795         addInstance instDeclName AttributeKind.global (evalPrio! default)
6796     if genSizeOfSpec.get (← getOptions) then
6797       mkSizeOfSpecTheorems indInfo.all.toArray fns recMap
6798
6799 builtin_initialize
6800   registerTraceClass `Meta.sizeOf
6801
6802 end Lean.Meta
6803 // ::::::::::::::
6804 // SynthInstance.lean
6805 // ::::::::::::::
6806 /-
6807 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
6808 Released under Apache 2.0 license as described in the file LICENSE.
6809 Authors: Daniel Selsam, Leonardo de Moura
6810
6811 Type class instance synthesizer using tabled resolution.
6812 -/
6813 import Lean.Meta.Basic
6814 import Lean.Meta.Instances
6815 import Lean.Meta.LevelDefEq
6816 import Lean.Meta.AbstractMVars
6817 import Lean.Meta.WHNF
6818 import Lean.Util.Profile
6819
6820 namespace Lean.Meta
6821
6822 register_builtin_option synthInstance.maxHeartbeats : Nat := {

```

```

6823   defValue := 500
6824   descr := "maximum amount of heartbeats per typeclass resolution problem. A heartbeat is number of (small) memory allocations (in
thousands), 0 means no limit"
6825 }
6826
6827 register_builtin_option synthInstance.maxSize : Nat := {
6828   defValue := 128
6829   descr := "maximum number of instances used to construct a solution in the type class instance synthesis procedure"
6830 }
6831
6832 namespace SynthInstance
6833
6834 def getMaxHeartbeats (opts : Options) : Nat :=
6835   synthInstance.maxHeartbeats.get opts * 1000
6836
6837 open Std (HashMap)
6838
6839 builtin_initialize inferTCGoalsRLAttr : TagAttribute ←
6840   registerTagAttribute `inferTCGoalsRL "instruct type class resolution procedure to solve goals from right to left for this instance"
6841
6842 def hasInferTCGoalsRLAttribute (env : Environment) (constName : Name) : Bool :=
6843   inferTCGoalsRLAttr.hasTag env constName
6844
6845 structure GeneratorNode where
6846   mvar      : Expr
6847   key       : Expr
6848   mctx      : MetavarContext
6849   instances : Array Expr
6850   currInstanceId : Nat
6851   deriving Inhabited
6852
6853 structure ConsumerNode where
6854   mvar      : Expr
6855   key       : Expr
6856   mctx      : MetavarContext
6857   subgoals  : List Expr
6858   size      : Nat -- instance size so far
6859   deriving Inhabited
6860
6861 inductive Waiter where
6862   | consumerNode : ConsumerNode → Waiter
6863   | root         : Waiter
6864
6865 def Waiter.isRoot : Waiter → Bool
6866   | Waiter.consumerNode _ => false
6867   | Waiter.root           => true
6868

```

```

6869 /-
6870   In tabled resolution, we creating a mapping from goals (e.g., `Coe Nat ?x`) to
6871   answers and waiters. Waiters are consumer nodes that are waiting for answers for a
6872   particular node.
6873
6874   We implement this mapping using a `HashMap` where the keys are
6875   normalized expressions. That is, we replace assignable metavariables
6876   with auxiliary free variables of the form `_tc.<idx>`. We do
6877   not declare these free variables in any local context, and we should
6878   view them as "normalized names" for metavariables. For example, the
6879   term `f ?m ?m ?n` is normalized as
6880   `f _tc.0 _tc.0 _tc.1`.
6881
6882   This approach is structural, and we may visit the same goal more
6883   than once if the different occurrences are just definitionally
6884   equal, but not structurally equal.
6885
6886   Remark: a metavariable is assignable only if its depth is equal to
6887   the metavar context depth.
6888 -/
6889 namespace MkTableKey
6890
6891 structure State where
6892   nextIdx : Nat := 0
6893   lmap    : HashMap MVarId Level := {}
6894   emap    : HashMap MVarId Expr := {}
6895
6896 abbrev M := ReaderT MetavarContext (StateM State)
6897
6898 partial def normLevel (u : Level) : M Level := do
6899   if !u.hasMVar then
6900     pure u
6901   else match u with
6902   | Level.succ v _ => return u.updateSucc! (← normLevel v)
6903   | Level.max v w _ => return u.updateMax! (← normLevel v) (← normLevel w)
6904   | Level.imax v w _ => return u.updateIMax! (← normLevel v) (← normLevel w)
6905   | Level.mvar mvarId _ =>
6906     let mctx ← read
6907     if !mctx.isLevelAssignable mvarId then
6908       pure u
6909     else
6910       let s ← get
6911       match s.lmap.find? mvarId with
6912       | some u' => pure u'
6913       | none    =>
6914         let u' := mkLevelParam $ Name.mkNum `_tc s.nextIdx
6915         modify fun s => { s with nextIdx := s.nextIdx + 1, lmap := s.lmap.insert mvarId u' }

```

```

6916         pure u'
6917     | u => pure u
6918
6919 partial def normExpr (e : Expr) : M Expr := do
6920   if !e.hasMVar then
6921     pure e
6922   else match e with
6923   | Expr.const _ us _ => return e.updateConst! (← us.mapM normLevel)
6924   | Expr.sort u _ => return e.updateSort! (← normLevel u)
6925   | Expr.app f a _ => return e.updateApp! (← normExpr f) (← normExpr a)
6926   | Expr.letE _ t v b _ => return e.updateLet! (← normExpr t) (← normExpr v) (← normExpr b)
6927   | Expr.forallE _ d b _ => return e.updateForallE! (← normExpr d) (← normExpr b)
6928   | Expr.lam _ d b _ => return e.updateLambdaE! (← normExpr d) (← normExpr b)
6929   | Expr.mdata _ b _ => return e.updateMData! (← normExpr b)
6930   | Expr.proj _ _ b _ => return e.updateProj! (← normExpr b)
6931   | Expr.mvar mvarId _ =>
6932     let mctx ← read
6933     if !mctx.isExprAssignable mvarId then
6934       pure e
6935     else
6936       let s ← get
6937       match s.emap.find? mvarId with
6938       | some e' => pure e'
6939       | none => do
6940         let e' := mkFVar $ Name.mkNum `_tc s.nextIdx
6941         modify fun s => { s with nextIdx := s.nextIdx + 1, emap := s.emap.insert mvarId e' }
6942         pure e'
6943   | _ => pure e
6944
6945 end MkTableKey
6946
6947 /- Remark: `mkTableKey` assumes `e` does not contain assigned metavariables. -/
6948 def mkTableKey (mctx : MetavarContext) (e : Expr) : Expr :=
6949   MkTableKey.normExpr e mctx |>.run' {}
6950
6951 structure Answer where
6952   result      : AbstractMVarsResult
6953   resultType  : Expr
6954   size       : Nat
6955
6956 instance : Inhabited Answer where
6957   default := { result := arbitrary, resultType := arbitrary, size := 0 }
6958
6959 structure TableEntry where
6960   waiters : Array Waiter
6961   answers : Array Answer := #[]
6962

```

```

6963 structure Context where
6964   maxResultSize : Nat
6965   maxHeartbeats : Nat
6966
6967 /-
6968   Remark: the SynthInstance.State is not really an extension of `Meta.State`.
6969   The field `postponed` is not needed, and the field `mctx` is misleading since
6970   `synthInstance` methods operate over different `MetavarContext`s simultaneously.
6971   That being said, we still use `extends` because it makes it simpler to move from
6972   `M` to `MetaM`.
6973 -/
6974 structure State where
6975   result      : Option Expr          := none
6976   generatorStack : Array GeneratorNode := #[]
6977   resumeStack  : Array (ConsumerNode × Answer) := #[]
6978   tableEntries : HashMap Expr TableEntry   := {}
6979
6980 abbrev SynthM := ReaderT Context $ StateRefT State MetaM
6981
6982 def checkMaxHeartbeats : SynthM Unit := do
6983   Core.checkMaxHeartbeatsCore "typeclass" `synthInstance.maxHeartbeats (← read).maxHeartbeats
6984
6985 @[inline] def mapMetaM (f : forall {α}, MetaM α → MetaM α) {α} : SynthM α → SynthM α :=
6986   monadMap @f
6987
6988 instance {α} : Inhabited (SynthM α) where
6989   default := fun _ _ => arbitrary
6990
6991 /-- Return globals and locals instances that may unify with `type` -/
6992 def getInstances (type : Expr) : MetaM (Array Expr) := do
6993   -- We must retrieve `localInstances` before we use `forallTelescopeReducing` because it will update the set of local instances
6994   let localInstances ← getLocalInstances
6995   forallTelescopeReducing type fun _ type => do
6996     let className? ← isClass? type
6997     match className? with
6998     | none    => throwError $ "type class instance expected" ++ indentExpr type
6999     | some className =>
7000       let globalInstances ← getGlobalInstancesIndex
7001       let result ← globalInstances.getUnify type
7002       -- Using insertion sort because it is stable and the array `result` should be mostly sorted.
7003       -- Most instances have default priority.
7004       let result := result.insertionSort fun e1 e2 => e1.priority < e2.priority
7005       let result ← result.mapM fun e => match e.val with
7006       | Expr.const constName us _ => return e.val.updateConst! (← us.mapM (fun _ => mkFreshLevelMVar))
7007       | _ => panic! "global instance is not a constant"
7008       trace[Meta.synthInstance.globalInstances]! "{type}, {result}"
7009       let result := localInstances.foldl (init := result) fun (result : Array Expr) linst =>

```

```

7010     if linst.className == className then result.push linst.fvar else result
7011     pure result
7012
7013 def mkGeneratorNode? (key mvar : Expr) : MetaM (Option GeneratorNode) := do
7014   let mvarType ← inferType mvar
7015   let mvarType ← instantiateMVars mvarType
7016   let instances ← getInstances mvarType
7017   if instances.isEmpty then
7018     pure none
7019   else
7020     let mctx ← getMCtx
7021     pure $ some {
7022       mvar      := mvar,
7023       key       := key,
7024       mctx      := mctx,
7025       instances := instances,
7026       currInstanceIdx := instances.size
7027     }
7028
7029 /-- Create a new generator node for `mvar` and add `waiter` as its waiter.
7030 `key` must be `mkTableKey mctx mvarType`. -/
7031 def newSubgoal (mctx : MetavarContext) (key : Expr) (mvar : Expr) (waiter : Waiter) : SynthM Unit :=
7032   withMCtx mctx do
7033     trace[Meta.synthInstance.newSubgoal]! key
7034     match (← mkGeneratorNode? key mvar) with
7035     | none      => pure ()
7036     | some node =>
7037       let entry : TableEntry := { waiters := #[waiter] }
7038       modify fun s =>
7039         { s with
7040           generatorStack := s.generatorStack.push node,
7041           tableEntries   := s.tableEntries.insert key entry }
7042
7043 def findEntry? (key : Expr) : SynthM (Option TableEntry) := do
7044   return (← get).tableEntries.find? key
7045
7046 def getEntry (key : Expr) : SynthM TableEntry := do
7047   match (← findEntry? key) with
7048   | none      => panic! "invalid key at synthInstance"
7049   | some entry => pure entry
7050
7051 /--
7052   Create a `key` for the goal associated with the given metavariable.
7053   That is, we create a key for the type of the metavariable.
7054
7055   We must instantiate assigned metavariables before we invoke `mkTableKey`. -/
7056 def mkTableKeyFor (mctx : MetavarContext) (mvar : Expr) : SynthM Expr :=

```

```

7057 withMCtx mctx do
7058   let mvarType ← inferType mvar
7059   let mvarType ← instantiateMVars mvarType
7060   return mkTableKey mctx mvarType
7061
7062 /- See `getSubgoals` and `getSubgoalsAux`
7063
7064   We use the parameter `j` to reduce the number of `instantiate*` invocations.
7065   It is the same approach we use at `forallTelescope` and `lambdaTelescope`.
7066   Given `getSubgoalsAux args j subgoals instVal type`,
7067   we have that `type.instantiateRevRange j args.size args` does not have loose bound variables. -/
7068 structure SubgoalsResult where
7069   subgoals      : List Expr
7070   instVal       : Expr
7071   instTypeBody  : Expr
7072
7073 private partial def getSubgoalsAux (lctx : LocalContext) (localInsts : LocalInstances) (xs : Array Expr)
7074   : Array Expr → Nat → List Expr → Expr → Expr → MetaM SubgoalsResult
7075 | args, j, subgoals, instVal, Expr.forallE n d b c => do
7076   let d      := d.instantiateRevRange j args.size args
7077   let mvarType ← mkForallFVars xs d
7078   let mvar     ← mkFreshExprMVarAt lctx localInsts mvarType
7079   let arg      := mkAppN mvar xs
7080   let instVal  := mkApp instVal arg
7081   let subgoals := if c.binderInfo.isInstImplicit then mvar::subgoals else subgoals
7082   let args     := args.push (mkAppN mvar xs)
7083   getSubgoalsAux lctx localInsts xs args j subgoals instVal b
7084 | args, j, subgoals, instVal, type => do
7085   let type := type.instantiateRevRange j args.size args
7086   let type ← whnf type
7087   if type.isForall then
7088     getSubgoalsAux lctx localInsts xs args args.size subgoals instVal type
7089   else
7090     pure {subgoals, instVal, type}
7091
7092 /--
7093   `getSubgoals lctx localInsts xs inst` creates the subgoals for the instance `inst`.
7094   The subgoals are in the context of the free variables `xs`, and
7095   `(lctx, localInsts)` is the local context and instances before we added the free variables to it.
7096
7097   This extra complication is required because
7098   1- We want all metavariables created by `synthInstance` to share the same local context.
7099   2- We want to ensure that applications such as `mvar xs` are higher order patterns.
7100
7101   The method `getGoals` create a new metavariable for each parameter of `inst`.
7102   For example, suppose the type of `inst` is `forall (x_1 : A_1) ... (x_n : A_n), B x_1 ... x_n`.
7103   Then, we create the metavariables `?m_i : forall xs, A_i`, and return the subset of these

```



```

7104 metavariables that are instance implicit arguments, and the expressions:
7105   - `inst (?m_1 xs) ... (?m_n xs)` (aka `instVal`)
7106   - `B (?m_1 xs) ... (?m_n xs)` -/
7107 def getSubgoals (lctx : LocalContext) (localInsts : LocalInstances) (xs : Array Expr) (inst : Expr) : MetaM SubgoalsResult := do
7108   let instType ← inferType inst
7109   let result ← getSubgoalsAux lctx localInsts xs #[] 0 [] inst instType
7110   match inst.getAppFn with
7111   | Expr.const constName _ _ =>
7112     let env ← getEnv
7113     if hasInferTCGoalsRLAttribute env constName then
7114       pure result
7115     else
7116       pure { result with subgoals := result.subgoals.reverse }
7117   | _ => pure result
7118
7119 def tryResolveCore (mvar : Expr) (inst : Expr) : MetaM (Option (MetavarContext × List Expr)) := do
7120   let mvarType ← inferType mvar
7121   let lctx ← getLCtx
7122   let localInsts ← getLocalInstances
7123   forallTelescopeReducing mvarType fun xs mvarTypeBody => do
7124     let (subgoals, instVal, instTypeBody) ← getSubgoals lctx localInsts xs inst
7125     trace[Meta.synthInstance.tryResolve]! "{mvarTypeBody} =?= {instTypeBody}"
7126     if (← isDefEq mvarTypeBody instTypeBody) then
7127       let instVal ← mkLambdaFVars xs instVal
7128       if (← isDefEq mvar instVal) then
7129         trace[Meta.synthInstance.tryResolve]! "success"
7130         pure (some ((← getMCtx), subgoals))
7131       else
7132         trace[Meta.synthInstance.tryResolve]! "failure assigning"
7133         pure none
7134     else
7135       trace[Meta.synthInstance.tryResolve]! "failure"
7136       pure none
7137
7138 /--
7139 Try to synthesize metavariable `mvar` using the instance `inst`.
7140 Remark: `mctx` contains `mvar`.
7141 If it succeeds, the result is a new updated metavariable context and a new list of subgoals.
7142 A subgoal is created for each instance implicit parameter of `inst`. -/
7143 def tryResolve (mctx : MetavarContext) (mvar : Expr) (inst : Expr) : SynthM (Option (MetavarContext × List Expr)) :=
7144   traceCtx `Meta.synthInstance.tryResolve <| withMCtx mctx <| tryResolveCore mvar inst
7145
7146 /--
7147 Assign a precomputed answer to `mvar`.
7148 If it succeeds, the result is a new updated metavariable context and a new list of subgoals. -/
7149 def tryAnswer (mctx : MetavarContext) (mvar : Expr) (answer : Answer) : SynthM (Option MetavarContext) :=
7150   withMCtx mctx do

```

```

7151   let (_, _, val) ← openAbstractMVarsResult answer.result
7152   if (← isDefEq mvar val) then
7153     pure (some (← getMCtx))
7154   else
7155     pure none
7156
7157 /-- Move waiters that are waiting for the given answer to the resume stack. -/
7158 def wakeUp (answer : Answer) : Waiter → SynthM Unit
7159 | Waiter.root      => do
7160   if answer.result.paramNames.isEmpty && answer.result.numMVars == 0 then
7161     modify fun s => { s with result := answer.result.expr }
7162   else
7163     let (_, _, answerExpr) ← openAbstractMVarsResult answer.result
7164     trace[Meta.synthInstance]! "skip answer containing metavariables {answerExpr}"
7165     pure ()
7166 | Waiter.consumerNode cNode =>
7167   modify fun s => { s with resumeStack := s.resumeStack.push (cNode, answer) }
7168
7169 def isNewAnswer (oldAnswers : Array Answer) (answer : Answer) : Bool :=
7170   oldAnswers.all fun oldAnswer => do
7171     -- Remark: isDefEq here is too expensive. TODO: if `==` is too imprecise, add some light normalization to `resultType` at
7172     -- `addAnswer`
7173     -- iseq ← isDefEq oldAnswer.resultType answer.resultType; pure (!iseq)
7174     oldAnswer.resultType != answer.resultType
7175
7176 private def mkAnswer (cNode : ConsumerNode) : MetaM Answer :=
7177   withMCtx cNode.mctx do
7178     traceM `Meta.synthInstance.newAnswer do m!"size: {cNode.size}, {← inferType cNode.mvar}"
7179     let val ← instantiateMVars cNode.mvar
7180     trace[Meta.synthInstance.newAnswer]! "val: {val}"
7181     let result ← abstractMVars val -- assignable metavariables become parameters
7182     let resultType ← inferType result.expr
7183     pure { result := result, resultType := resultType, size := cNode.size + 1 }
7184 /--
7185   Create a new answer after `cNode` resolved all subgoals.
7186   That is, `cNode.subgoals == []`.
7187   And then, store it in the tabled entries map, and wakeup waiters. -/
7188 def addAnswer (cNode : ConsumerNode) : SynthM Unit := do
7189   if cNode.size ≥ (← read).maxResultSize then
7190     traceM `Meta.synthInstance.discarded do m!"size: {cNode.size} ≥ {(← read).maxResultSize}, {← inferType cNode.mvar}"
7191     return ()
7192   else
7193     let answer ← mkAnswer cNode
7194     -- Remark: `answer` does not contain assignable or assigned metavariables.
7195     let key := cNode.key
7196     let entry ← getEntry key

```

```

7197     if isNewAnswer entry.answers answer then
7198         let newEntry := { entry with answers := entry.answers.push answer }
7199         modify fun s => { s with tableEntries := s.tableEntries.insert key newEntry }
7200         entry.waiters.forM (wakeUp answer)
7201
7202 /-- Process the next subgoal in the given consumer node. -/
7203 def consume (cNode : ConsumerNode) : SynthM Unit :=
7204     match cNode.subgoals with
7205     | []      => addAnswer cNode
7206     | mvar::_ => do
7207         let waiter := Waiter.consumerNode cNode
7208         let key ← mkTableKeyFor cNode.mctx mvar
7209         let entry? ← findEntry? key
7210         match entry? with
7211         | none      => newSubgoal cNode.mctx key mvar waiter
7212         | some entry => modify fun s =>
7213             { s with
7214                 resumeStack := entry.answers.foldl (fun s answer => s.push (cNode, answer)) s.resumeStack,
7215                 tableEntries := s.tableEntries.insert key { entry with waiters := entry.waiters.push waiter } }
7216
7217 def getTop : SynthM GeneratorNode := do
7218     pure (← get).generatorStack.back
7219
7220 @[inline] def modifyTop (f : GeneratorNode → GeneratorNode) : SynthM Unit :=
7221     modify fun s => { s with generatorStack := s.generatorStack.modify (s.generatorStack.size - 1) f }
7222
7223 /-- Try the next instance in the node on the top of the generator stack. -/
7224 def generate : SynthM Unit := do
7225     let gNode ← getTop
7226     if gNode.currInstanceId == 0 then
7227         modify fun s => { s with generatorStack := s.generatorStack.pop }
7228     else do
7229         let key := gNode.key
7230         let idx := gNode.currInstanceId - 1
7231         let inst := gNode.instances.get! idx
7232         let mctx := gNode.mctx
7233         let mvar := gNode.mvar
7234         trace[Meta.synthInstance.generate]! "instance {inst}"
7235         modifyTop fun gNode => { gNode with currInstanceId := idx }
7236         match (← tryResolve mctx mvar inst) with
7237         | none      => pure ()
7238         | some (mctx, subgoals) => consume { key := key, mvar := mvar, subgoals := subgoals, mctx := mctx, size := 0 }
7239
7240 def getNextToResume : SynthM (ConsumerNode × Answer) := do
7241     let s ← get
7242     let r := s.resumeStack.back
7243     modify fun s => { s with resumeStack := s.resumeStack.pop }

```

```

7244 pure r
7245
7246 /--
7247   Given `(cNode, answer)` on the top of the resume stack, continue execution by using `answer` to solve the
7248   next subgoal. -/
7249 def resume : SynthM Unit := do
7250   let (cNode, answer) ← getNextToResume
7251   match cNode.subgoals with
7252   | []      => panic! "resume found no remaining subgoals"
7253   | mvar::rest =>
7254     match (← tryAnswer cNode.mctx mvar answer) with
7255     | none      => pure ()
7256     | some mctx =>
7257       withMCtx mctx <| traceM `Meta.synthInstance.resume do
7258         let goal   ← inferType cNode.mvar
7259         let subgoal ← inferType mvar
7260         pure m!"size: {cNode.size + answer.size}, {goal} <== {subgoal}"
7261         consume { key := cNode.key, mvar := cNode.mvar, subgoals := rest, mctx := mctx, size := cNode.size + answer.size }
7262
7263 def step : SynthM Bool := do
7264   checkMaxHeartbeats
7265   let s ← get
7266   if !s.resumeStack.isEmpty then
7267     resume
7268     pure true
7269   else if !s.generatorStack.isEmpty then
7270     generate
7271     pure true
7272   else
7273     pure false
7274
7275 def getResult : SynthM (Option Expr) := do
7276   pure (← get).result
7277
7278 partial def synth : SynthM (Option Expr) := do
7279   if (← step) then
7280     match (← getResult) with
7281     | none      => synth
7282     | some result => pure result
7283   else
7284     trace[Meta.synthInstance]! "failed"
7285     pure none
7286
7287 def main (type : Expr) (maxResultSize : Nat) : MetaM (Option Expr) :=
7288   withCurrHeartbeats <| traceCtx `Meta.synthInstance do
7289     trace[Meta.synthInstance]! "main goal {type}"
7290     let mvar ← mkFreshExprMVar type

```

```

7291     let mctx ← getMCtx
7292     let key   := mkTableKey mctx type
7293     let action : SynthM (Option Expr) := do
7294       newSubgoal mctx key mvar Waiter.root
7295       synth
7296     action.run { maxResultSize := maxResultSize, maxHeartbeats := getMaxHeartbeats (← getOptions) } |>.run' {}
7297
7298 end SynthInstance
7299
7300 /-
7301 Type class parameters can be annotated with `outParam` annotations.
7302
7303 Given `C a_1 ... a_n`, we replace `a_i` with a fresh metavariable `?m_i` IF
7304 `a_i` is an `outParam`.
7305 The result is type correct because we reject type class declarations IF
7306 it contains a regular parameter X that depends on an `out` parameter Y.
7307
7308 Then, we execute type class resolution as usual.
7309 If it succeeds, and metavariables ?m_i have been assigned, we try to unify
7310 the original type `C a_1 ... a_n` with the normalized one.
7311 -/
7312
7313 private def preprocess (type : Expr) : MetaM Expr :=
7314   forallTelescopeReducing type fun xs type => do
7315     let type ← whnf type
7316     mkForallFVars xs type
7317
7318 private def preprocessLevels (us : List Level) : MetaM (List Level × Bool) := do
7319   let mut r := #[]
7320   let mut modified := false
7321   for u in us do
7322     let u ← instantiateLevelMVars u
7323     if u.hasMVar then
7324       r := r.push (← mkFreshLevelMVar)
7325       modified := true
7326     else
7327       r := r.push u
7328   return (r.toList, modified)
7329
7330 private partial def preprocessArgs (type : Expr) (i : Nat) (args : Array Expr) : MetaM (Array Expr) := do
7331   if h : i < args.size then
7332     let type ← whnf type
7333     match type with
7334     | Expr.forallE _ d b _ => do
7335       let arg := args.get (i, h)
7336       let arg ← if isOutParam d then mkFreshExprMVar d else pure arg
7337       let args := args.set (i, h) arg

```

```

7338     preprocessArgs (b.instantiate1 arg) (i+1) args
7339   | _ =>
7340     throwError "type class resolution failed, insufficient number of arguments" -- TODO improve error message
7341   else
7342     return args
7343
7344 private def preprocessOutParam (type : Expr) : MetaM Expr :=
7345   forallTelescope type fun xs typeBody => do
7346     match typeBody.getAppFn with
7347     | c@(Expr.const constName us _) =>
7348       let env ← getEnv
7349       if !hasOutParams env constName then
7350         /- We treat all universe level parameters as "outParam" -/
7351         let (us, modified) ← preprocessLevels us
7352         if modified then
7353           let c := mkConst constName us
7354           mkForallFVars xs (mkAppN c typeBody.getAppArgs)
7355         else
7356           return type
7357     else do
7358       let args := typeBody.getAppArgs
7359       let (us, _) ← preprocessLevels us
7360       let c := mkConst constName us
7361       let cType ← inferType c
7362       let args ← preprocessArgs cType 0 args
7363       mkForallFVars xs (mkAppN c args)
7364   | _ =>
7365     return type
7366
7367 /-
7368 Remark: when `maxResultSize? == none`, the configuration option `synthInstance.maxResultSize` is used.
7369 Remark: we use a different option for controlling the maximum result size for coercions.
7370 -/
7371
7372 def synthInstance? (type : Expr) (maxResultSize? : Option Nat := none) : MetaM (Option Expr) := do profileitM Exception "typeclass
inference" (← getOptions) do
7373   let opts ← getOptions
7374   let maxResultSize := maxResultSize?.getD (synthInstance.maxSize.get opts)
7375   let inputConfig ← getConfig
7376   withConfig (fun config => { config with isDefEqStuckEx := true, transparency := TransparencyMode.instances,
7377                                     foApprox := true, ctxApprox := true, constApprox := false }) do
7378     let type ← instantiateMVars type
7379     let type ← preprocess type
7380     let s ← get
7381     match s.cache.synthInstance.find? type with
7382     | some result => pure result
7383     | none       =>

```

```

7384 let result? ← withNewMCtxDepth do
7385   let normType ← preprocessOutParam type
7386   trace[Meta.synthInstance]! "{type} ==> {normType}"
7387   match (← SynthInstance.main normType maxResultSize) with
7388   | none      => pure none
7389   | some result =>
7390     trace[Meta.synthInstance]! "FOUND result {result}"
7391     let result ← instantiateMVars result
7392     if (← hasAssignableMVar result) then
7393       trace[Meta.synthInstance]! "Failed has assignable mvar {result.setOption `pp.all true}"
7394       pure none
7395     else
7396       pure (some result)
7397 let result? ← match result? with
7398 | none      => pure none
7399 | some result => do
7400   trace[Meta.synthInstance]! "result {result}"
7401   let resultType ← inferType result
7402   if (← withConfig (fun _ => inputConfig) <| isDefEq type resultType) then
7403     let result ← instantiateMVars result
7404     pure (some result)
7405   else
7406     trace[Meta.synthInstance]! "result type{indentExpr resultType}\nis not definitionally equal to{indentExpr type}"
7407     pure none
7408 if type.hasMVar then
7409   pure result?
7410 else do
7411   modify fun s => { s with cache := { s.cache with synthInstance := s.cache.synthInstance.insert type result? } }
7412   pure result?
7413
7414 /--
7415   Return `LOption.some r` if succeeded, `LOption.none` if it failed, and `LOption.undef` if
7416   instance cannot be synthesized right now because `type` contains metavariables. -/
7417 def trySynthInstance (type : Expr) (maxResultSize? : Option Nat := none) : MetaM (LOption Expr) := do
7418   catchInternalId isDefEqStuckExceptionId
7419   (toLOptionM <| synthInstance? type maxResultSize?)
7420   (fun _ => pure LOption.undef)
7421
7422 def synthInstance (type : Expr) (maxResultSize? : Option Nat := none) : MetaM Expr :=
7423   catchInternalId isDefEqStuckExceptionId
7424   (do
7425     let result? ← synthInstance? type maxResultSize?
7426     match result? with
7427     | some result => pure result
7428     | none      => throwError! "failed to synthesize{indentExpr type}")
7429   (fun _ => throwError! "failed to synthesize{indentExpr type}")
7430

```

```

7431 private def synthPendingImp (mvarId : MVarId) (maxResultSize? : Option Nat) : MetaM Bool := do
7432   let mvarDecl ← getMVarDecl mvarId
7433   match mvarDecl.kind with
7434   | MetavarKind.synthetic =>
7435     match (← isClass? mvarDecl.type) with
7436     | none    => pure false
7437     | some _ => do
7438       let val? ← catchInternalId isDefEqStuckExceptionId (synthInstance? mvarDecl.type maxResultSize?) (fun _ => pure none)
7439       match val? with
7440       | none    => pure false
7441       | some val =>
7442         if (← isExprMVarAssigned mvarId) then
7443           pure false
7444         else
7445           assignExprMVar mvarId val
7446           pure true
7447   | _ => pure false
7448
7449 builtin_initialize
7450   synthPendingRef.set (synthPendingImp · none)
7451
7452 builtin_initialize
7453   registerTraceClass `Meta.synthInstance
7454   registerTraceClass `Meta.synthInstance.globalInstances
7455   registerTraceClass `Meta.synthInstance.newSubgoal
7456   registerTraceClass `Meta.synthInstance.tryResolve
7457   registerTraceClass `Meta.synthInstance.resume
7458   registerTraceClass `Meta.synthInstance.generate
7459
7460 end Lean.Meta
7461 // ::::::::::::::
7462 // Tactic.lean
7463 // ::::::::::::::
7464 /-
7465 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
7466 Released under Apache 2.0 license as described in the file LICENSE.
7467 Authors: Leonardo de Moura
7468 -/
7469 import Lean.Meta.Tactic.Intro
7470 import Lean.Meta.Tactic.Assumption
7471 import Lean.Meta.Tactic.Contradiction
7472 import Lean.Meta.Tactic.Apply
7473 import Lean.Meta.Tactic.Revert
7474 import Lean.Meta.Tactic.Clear
7475 import Lean.Meta.Tactic.Assert
7476 import Lean.Meta.Tactic.Rewrite
7477 import Lean.Meta.Tactic.Generalize

```



```

7478 import Lean.Meta.Tactic.Replace
7479 import Lean.Meta.Tactic.Induction
7480 import Lean.Meta.Tactic.Cases
7481 import Lean.Meta.Tactic.ElimInfo
7482 import Lean.Meta.Tactic.Delta
7483 import Lean.Meta.Tactic.Constructor
7484 import Lean.Meta.Tactic.Simp
7485 // ::::::::::::::
7486 // Transform.lean
7487 // ::::::::::::::
7488 /-
7489 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
7490 Released under Apache 2.0 license as described in the file LICENSE.
7491 Authors: Leonardo de Moura
7492 -/
7493 import Lean.Meta.Basic
7494
7495 namespace Lean
7496
7497 inductive TransformStep where
7498   | done (e : Expr)
7499   | visit (e : Expr)
7500
7501 namespace Core
7502
7503 /--
7504   Transform the expression `input` using `pre` and `post`.
7505   - `pre s` is invoked before visiting the children of subterm `s`. If the result is `TransformStep.visit sNew`, then
7506     `sNew` is traversed by transform. If the result is `TransformStep.visit sNew`, then `s` is just replaced with `sNew`.
7507     In both cases, `sNew` must be definitionally equal to `s`
7508   - `post s` is invoked after visiting the children of subterm `s`.
7509
7510   The term `s` in both `pre s` and `post s` may contain loose bound variables. So, this method is not appropriate for
7511   if one needs to apply operations (e.g., `whnf`, `inferType`) that do not handle loose bound variables.
7512   Consider using `Meta.transform` to avoid loose bound variables.
7513
7514   This method is useful for applying transformations such as beta-reduction and delta-reduction.
7515 -/
7516 partial def transform {m} [Monad m] [MonadLiftT CoreM m] [MonadControlT CoreM m]
7517   (input : Expr)
7518   (pre   : Expr → m TransformStep := fun e => return TransformStep.visit e)
7519   (post  : Expr → m TransformStep := fun e => return TransformStep.done e)
7520   : m Expr :=
7521   let inst : STWorld IO.RealWorld m := {}
7522   let inst : MonadLiftT (ST IO.RealWorld) m := { monadLift := fun x => liftM (m := CoreM) (liftM (m := ST IO.RealWorld) x) }
7523   let rec visit (e : Expr) : MonadCacheT Expr Expr m Expr :=
7524     checkCache e fun _ => Core.withIncRecDepth do

```

```

7525     let rec visitPost (e : Expr) : MonadCacheT Expr Expr m Expr := do
7526         match (← post e) with
7527         | TransformStep.done e => pure e
7528         | TransformStep.visit e => visit e
7529     match (← pre e) with
7530     | TransformStep.done e => pure e
7531     | TransformStep.visit e => match e with
7532     | Expr.forallE _ d b _ => visitPost (e.updateForallE! (← visit d) (← visit b))
7533     | Expr.lam _ d b _ => visitPost (e.updateLambdaE! (← visit d) (← visit b))
7534     | Expr.letE _ t v b _ => visitPost (e.updateLet! (← visit t) (← visit v) (← visit b))
7535     | Expr.app .. => e.withApp fun f args => do visitPost (mkAppN (← visit f) (← args.mapM visit))
7536     | Expr.mdata _ b _ => visitPost (e.updateMData! (← visit b))
7537     | Expr.proj _ _ b _ => visitPost (e.updateProj! (← visit b))
7538     | _ => visitPost e
7539     visit input |>.run
7540
7541 def betaReduce (e : Expr) : CoreM Expr :=
7542     transform e (pre := fun e => return TransformStep.visit e.headBeta)
7543
7544 end Core
7545
7546 namespace Meta
7547
7548 /--
7549     Similar to `Core.transform`, but terms provided to `pre` and `post` do not contain loose bound variables.
7550     So, it is safe to use any `MetaM` method at `pre` and `post`. -/
7551 partial def transform {m} [Monad m] [MonadLiftT MetaM m] [MonadControlT MetaM m]
7552     (input : Expr)
7553     (pre : Expr → m TransformStep := fun e => return TransformStep.visit e)
7554     (post : Expr → m TransformStep := fun e => return TransformStep.done e)
7555     : m Expr :=
7556     let inst : STWorld IO.RealWorld m := {}
7557     let inst : MonadLiftT (ST IO.RealWorld) m := { monadLift := fun x => liftM (m := MetaM) (liftM (m := ST IO.RealWorld) x) }
7558     let rec visit (e : Expr) : MonadCacheT Expr Expr m Expr :=
7559         checkCache e fun _ => Meta.withIncRecDepth do
7560             let rec visitPost (e : Expr) : MonadCacheT Expr Expr m Expr := do
7561                 match (← post e) with
7562                 | TransformStep.done e => pure e
7563                 | TransformStep.visit e => visit e
7564             let rec visitLambda (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
7565                 match e with
7566                 | Expr.lam n d b c =>
7567                     withLocalDecl n c.binderInfo (← visit (d.instantiateRev fvars)) fun x =>
7568                         visitLambda (fvars.push x) b
7569                 | e => visitPost (← mkLambdaFVars fvars (← visit (e.instantiateRev fvars)))
7570             let rec visitForall (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
7571                 match e with

```

```

7572 | Expr.forallE n d b c =>
7573   withLocalDecl n c.binderInfo (← visit (d.instantiateRev fvars)) fun x =>
7574     visitForall (fvars.push x) b
7575 | e => visitPost (← mkForallFVars fvars (← visit (e.instantiateRev fvars)))
7576 let rec visitLet (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
7577   match e with
7578   | Expr.letE n t v b _ =>
7579     withLetDecl n (← visit (t.instantiateRev fvars)) (← visit (v.instantiateRev fvars)) fun x =>
7580       visitLet (fvars.push x) b
7581   | e => visitPost (← mkLetFVars fvars (← visit (e.instantiateRev fvars)))
7582 let visitApp (e : Expr) : MonadCacheT Expr Expr m Expr :=
7583   e.withApp fun f args => do
7584     visitPost (mkAppN (← visit f) (← args.mapM visit))
7585 match (← pre e) with
7586 | TransformStep.done e => pure e
7587 | TransformStep.visit e => match e with
7588 | Expr.forallE ..      => visitForall #[] e
7589 | Expr.lam ..          => visitLambda #[] e
7590 | Expr.letE ..         => visitLet #[] e
7591 | Expr.app ..          => visitApp e
7592 | Expr.mdata _ b _     => visitPost (e.updateMData! (← visit b))
7593 | Expr.proj _ _ b _    => visitPost (e.updateProj! (← visit b))
7594 | _                    => visitPost e
7595 visit input |>.run
7596
7597 def zetaReduce (e : Expr) : MetaM Expr := do
7598   let lctx ← getLCtx
7599   let pre (e : Expr) : CoreM TransformStep := do
7600     match e with
7601     | Expr.fvar fvarId _ =>
7602       match lctx.find? fvarId with
7603       | none => return TransformStep.done e
7604       | some localDecl =>
7605         if let some value := localDecl.value? then
7606           return TransformStep.visit value
7607         else
7608           return TransformStep.done e
7609     | e => if e.hasFVar then return TransformStep.visit e else return TransformStep.done e
7610   liftM (m := CoreM) <| Core.transform e (pre := pre)
7611
7612 end Meta
7613 end Lean
7614 // ::::::::::::::
7615 // TransparencyMode.lean
7616 // ::::::::::::::
7617 /-
7618 Copyright (c) 2020 Microsoft Corporation. All rights reserved.

```

```

7619 Released under Apache 2.0 license as described in the file LICENSE.
7620 Authors: Leonardo de Moura
7621 -/
7622 namespace Lean.Meta
7623
7624 inductive TransparencyMode where
7625   | all | default | reducible | instances
7626   deriving Inhabited, BEq, Repr
7627
7628 namespace TransparencyMode
7629
7630 def hash : TransparencyMode → USize
7631   | all      => 7
7632   | default  => 11
7633   | reducible => 13
7634   | instances => 17
7635
7636 instance : Hashable TransparencyMode := {hash}
7637
7638 def lt : TransparencyMode → TransparencyMode → Bool
7639   | reducible, default  => true
7640   | reducible, all      => true
7641   | reducible, instances => true
7642   | instances, default  => true
7643   | instances, all      => true
7644   | default, all        => true
7645   | _, _                => false
7646
7647 end TransparencyMode
7648
7649 end Lean.Meta
7650 // ::::::::::::::
7651 // UnificationHint.lean
7652 // ::::::::::::::
7653 /-
7654 Copyright (c) 2020 Microsoft Corporation. All rights reserved.
7655 Released under Apache 2.0 license as described in the file LICENSE.
7656 Authors: Leonardo de Moura
7657 -/
7658 import Lean.ScopedEnvExtension
7659 import Lean.Util.Recognizers
7660 import Lean.Meta.DiscrTree
7661 import Lean.Meta.LevelDefEq
7662 import Lean.Meta.SynthInstance
7663
7664 namespace Lean.Meta
7665

```

```

7666 structure UnificationHintEntry where
7667   keys      : Array DiscrTree.Key
7668   val       : Name
7669   deriving Inhabited
7670
7671 structure UnificationHints where
7672   discrTree : DiscrTree Name := DiscrTree.empty
7673   deriving Inhabited
7674
7675 instance : ToFormat UnificationHints where
7676   format h := fmt h.dscrTree
7677
7678 def UnificationHints.add (hints : UnificationHints) (e : UnificationHintEntry) : UnificationHints :=
7679   { hints with discrTree := hints.dscrTree.insertCore e.keys e.val }
7680
7681 builtin_initialize unificationHintExtension : SimpleScopedEnvExtension UnificationHintEntry UnificationHints ←
7682   registerSimpleScopedEnvExtension {
7683     name      := `unifHints
7684     addEntry  := UnificationHints.add
7685     initial   := {}
7686   }
7687
7688 structure UnificationConstraint where
7689   lhs : Expr
7690   rhs : Expr
7691
7692 structure UnificationHint where
7693   pattern : UnificationConstraint
7694   constraints : List UnificationConstraint
7695
7696 private partial def decodeUnificationHint (e : Expr) : ExceptT MessageData Id UnificationHint := do
7697   decode e #[]
7698 where
7699   decodeConstraint (e : Expr) : ExceptT MessageData Id UnificationConstraint :=
7700     match e.eq? with
7701     | some (_, lhs, rhs) => return UnificationConstraint.mk lhs rhs
7702     | none => throw m!"invalid unification hint constraint, unexpected term{indentExpr e}"
7703   decode (e : Expr) (cs : Array UnificationConstraint) : ExceptT MessageData Id UnificationHint := do
7704     match e with
7705     | Expr.forallE _ d b _ => do
7706       let c ← decodeConstraint d
7707       if b.hasLooseBVars then
7708         throw m!"invalid unification hint constraint, unexpected dependency{indentExpr e}"
7709       decode b (cs.push c)
7710     | _ => do
7711       let p ← decodeConstraint e
7712       return { pattern := p, constraints := cs.toList }

```

```

7713
7714 private partial def validateHint (declName : Name) (hint : UnificationHint) : MetaM Unit := do
7715   hint.constraints.forM fun c => do
7716     unless (← isDefEq c.lhs c.rhs) do
7717       throwError! "invalid unification hint, failed to unify constraint left-hand-side{indentExpr c.lhs}\nwith right-hand-
side{indentExpr c.rhs}"
7718   unless (← isDefEq hint.pattern.lhs hint.pattern.rhs) do
7719     throwError! "invalid unification hint, failed to unify pattern left-hand-side{indentExpr hint.pattern.lhs}\nwith right-hand-
side{indentExpr hint.pattern.rhs}"
7720
7721 def addUnificationHint (declName : Name) (kind : AttributeKind) : MetaM Unit :=
7722   withNewMCtxDepth do
7723     let info ← getConstInfo declName
7724     match info.value? with
7725     | none => throwError! "invalid unification hint, it must be a definition"
7726     | some val =>
7727       let (_, _, body) ← lambdaMetaTelescope val
7728       match decodeUnificationHint body with
7729       | Except.error msg => throwError msg
7730       | Except.ok hint =>
7731         let keys ← DiscrTree.mkPath hint.pattern.lhs
7732         validateHint declName hint
7733         unificationHintExtension.add { keys := keys, val := declName } kind
7734         trace[Meta.debug]! "addUnificationHint: {unificationHintExtension.getState (← getEnv)}"
7735
7736 builtin_initialize
7737   registerBuiltinAttribute {
7738     name   := `unificationHint
7739     descr := "unification hint"
7740     add    := fun declName stx kind => do
7741       Attribute.Builtin.ensureNoArgs stx
7742       discard <| addUnificationHint declName kind |>.run
7743   }
7744
7745 def tryUnificationHints (t s : Expr) : MetaM Bool := do
7746   trace[Meta.isDefEq.hint]! "{t} == {s}"
7747   unless (← read).config.unificationHints do
7748     return false
7749   if t.isMVar then
7750     return false
7751   let hints := unificationHintExtension.getState (← getEnv)
7752   let candidates ← hints.dscrTree.getMatch t
7753   for candidate in candidates do
7754     if (← tryCandidate candidate) then
7755       return true
7756   return false
7757 where

```

```

7758 isDefEqPattern p e :=
7759   withReducible <| Meta.isExprDefEqAux p e
7760
7761 tryCandidate candidate : MetaM Bool :=
7762   traceCtx `Meta.isDefEq.hint <| commitWhen do
7763     trace[Meta.isDefEq.hint]! "trying hint {candidate} at {t} == {s}"
7764     let cinfo ← getConstInfo candidate
7765     let us ← cinfo.levelParams.mapM fun _ => mkFreshLevelMVar
7766     let val := cinfo.instantiateValueLevelParams us
7767     let (xs, bis, body) ← lambdaMetaTelescope val
7768     let hint? ← withConfig (fun cfg => { cfg with unificationHints := false }) do
7769       match decodeUnificationHint body with
7770       | Except.error _ => return none
7771       | Except.ok hint =>
7772         if (← isDefEqPattern hint.pattern.lhs t <&& isDefEqPattern hint.pattern.rhs s) then
7773           return some hint
7774         else
7775           return none
7776     match hint? with
7777     | none => return false
7778     | some hint =>
7779       trace[Meta.isDefEq.hint]! "{candidate} succeeded, applying constraints"
7780       for c in hint.constraints do
7781         unless (← Meta.isExprDefEqAux c.lhs c.rhs) do
7782           return false
7783       for x in xs, bi in bis do
7784         if bi == BinderInfo.instImplicit then
7785           match (← trySynthInstance (← inferType x)) with
7786           | LOption.some val => unless (← isDefEq x val) do return false
7787           | _ => return false
7788       return true
7789
7790 builtin_initialize
7791   registerTraceClass `Meta.isDefEq.hint
7792
7793 end Lean.Meta
7794 // ::::::::::::::
7795 // WHNF.lean
7796 // ::::::::::::::
7797 /-
7798 Copyright (c) 2019 Microsoft Corporation. All rights reserved.
7799 Released under Apache 2.0 license as described in the file LICENSE.
7800 Authors: Leonardo de Moura
7801 -/
7802 import Lean.ToExpr
7803 import Lean.AuxRecursor
7804 import Lean.ProjFns

```

```

7805 import Lean.Meta.Basic
7806 import Lean.Meta.LevelDefEq
7807 import Lean.Meta.GetConst
7808 import Lean.Meta.Match.MatcherInfo
7809
7810 namespace Lean.Meta
7811
7812 /- =====
7813   Smart unfolding support
7814   ===== -/
7815
7816 def smartUnfoldingSuffix := "_sunfold"
7817
7818 @[inline] def mkSmartUnfoldingNameFor (declName : Name) : Name :=
7819   Name.mkStr declName smartUnfoldingSuffix
7820
7821 register_builtin_option smartUnfolding : Bool := {
7822   defValue := true
7823   descr := "when computing weak head normal form, use auxiliary definition created for functions defined by structural recursion"
7824 }
7825
7826 /- =====
7827   Helper methods
7828   ===== -/
7829 def isAuxDef (constName : Name) : MetaM Bool := do
7830   let env ← getEnv
7831   return isAuxRecursor env constName || isNoConfusion env constName
7832
7833 @[inline] private def matchConstAux {α} (e : Expr) (failK : Unit → MetaM α) (k : ConstantInfo → List Level → MetaM α) : MetaM α :=
7834   match e with
7835   | Expr.const name lvls _ => do
7836     let (some cinfo) ← getConst? name | failK ()
7837     k cinfo lvls
7838   | _ => failK ()
7839
7840 /- =====
7841   Helper functions for reducing recursors
7842   ===== -/
7843
7844 private def getFirstCtor (d : Name) : MetaM (Option Name) := do
7845   let some (ConstantInfo.inductInfo { ctors := ctor::_, ..}) ← getConstNoEx? d | pure none
7846   return some ctor
7847
7848 private def mkNullaryCtor (type : Expr) (nparams : Nat) : MetaM (Option Expr) :=
7849   match type.getAppFn with
7850   | Expr.const d lvls _ => do
7851     let (some ctor) ← getFirstCtor d | pure none

```



```

7852     return mkAppN (mkConst ctor lvls) (type.getAppArgs.shrink nparams)
7853 | _ =>
7854     return none
7855
7856 def toCtorIfLit : Expr → Expr
7857 | Expr.lit (Literal.natVal v) _ =>
7858     if v == 0 then mkConst `Nat.zero
7859     else mkApp (mkConst `Nat.succ) (mkNatLit (v-1))
7860 | Expr.lit (Literal.strVal v) _ =>
7861     mkApp (mkConst `String.mk) (toExpr v.toList)
7862 | e => e
7863
7864 private def getRecRuleFor (recVal : RecursorVal) (major : Expr) : Option RecursorRule :=
7865     match major.getAppFn with
7866     | Expr.const fn _ _ => recVal.rules.find? fun r => r.ctor == fn
7867     | _                  => none
7868
7869 private def toCtorWhenK (recVal : RecursorVal) (major : Expr) : MetaM (Option Expr) := do
7870     let majorType ← inferType major
7871     let majorType ← whnf majorType
7872     let majorTypeI := majorType.getAppFn
7873     if !majorTypeI.isConstOf recVal.getInduct then
7874         return none
7875     else if majorType.hasExprMVar && majorType.getAppArgs[recVal.numParams:].any Expr.hasExprMVar then
7876         return none
7877     else do
7878         let (some newCtorApp) ← mkNullaryCtor majorType recVal.numParams | pure none
7879         let newType ← inferType newCtorApp
7880         if (← isDefEq majorType newType) then
7881             return newCtorApp
7882         else
7883             return none
7884
7885 /-- Auxiliary function for reducing recursor applications. -/
7886 private def reduceRec {α} (recVal : RecursorVal) (recLvls : List Level) (recArgs : Array Expr) (failK : Unit → MetaM α) (successK :
Expr → MetaM α) : MetaM α :=
7887     let majorIdx := recVal.getMajorIdx
7888     if h : majorIdx < recArgs.size then do
7889         let major := recArgs.get {majorIdx, h}
7890         let mut major ← whnf major
7891         if recVal.k then
7892             let newMajor ← toCtorWhenK recVal major
7893             major := newMajor.getD major
7894         let major := toCtorIfLit major
7895         match getRecRuleFor recVal major with
7896         | some rule =>
7897             let majorArgs := major.getAppArgs

```

```

7898   if recLvls.length != recVal.levelParams.length then
7899     failK ()
7900   else
7901     let rhs := rule.rhs.instantiateLevelParams recVal.levelParams recLvls
7902     -- Apply parameters, motives and minor premises from recursor application.
7903     let rhs := mkAppRange rhs 0 (recVal.numParams+recVal.numMotives+recVal.numMinors) recArgs
7904     /- The number of parameters in the constructor is not necessarily
7905        equal to the number of parameters in the recursor when we have
7906        nested inductive types. -/
7907     let nparams := majorArgs.size - rule.nfields
7908     let rhs := mkAppRange rhs nparams majorArgs.size majorArgs
7909     let rhs := mkAppRange rhs (majorIdx + 1) recArgs.size recArgs
7910     successK rhs
7911   | none => failK ()
7912 else
7913   failK ()
7914
7915 /- =====
7916   Helper functions for reducing Quot.lift and Quot.ind
7917   ===== -/
7918
7919 /- Auxiliary function for reducing `Quot.lift` and `Quot.ind` applications. -/
7920 private def reduceQuotRec {α} (recVal : QuotVal) (recLvls : List Level) (recArgs : Array Expr) (failK : Unit → MetaM α) (successK :
Expr → MetaM α) : MetaM α :=
7921   let process (majorPos argPos : Nat) : MetaM α :=
7922     if h : majorPos < recArgs.size then do
7923       let major := recArgs.get (majorPos, h)
7924       let major ← whnf major
7925       match major with
7926       | Expr.app (Expr.app (Expr.app (Expr.const majorFn _ _) _ _) _ _) majorArg _ => do
7927         let some (ConstantInfo.quotInfo { kind := QuotKind.ctor, .. }) ← getConstNoEx? majorFn | failK ()
7928         let f := recArgs[argPos]
7929         let r := mkApp f majorArg
7930         let recAry := majorPos + 1
7931         successK $ mkAppRange r recAry recArgs.size recArgs
7932       | _ => failK ()
7933     else
7934       failK ()
7935   match recVal.kind with
7936   | QuotKind.lift => process 5 3
7937   | QuotKind.ind  => process 4 3
7938   | _             => failK ()
7939
7940 /- =====
7941   Helper function for extracting "stuck term"
7942   ===== -/
7943

```

```

7944 mutual
7945   private partial def isRecStuck? (recVal : RecursorVal) (recLvls : List Level) (recArgs : Array Expr) : MetaM (Option MVarId) :=
7946     if recVal.k then
7947       -- TODO: improve this case
7948       return none
7949     else do
7950       let majorIdx := recVal.getMajorIdx
7951       if h : majorIdx < recArgs.size then do
7952         let major := recArgs.get (majorIdx, h)
7953         let major ← whnf major
7954         getStuckMVar? major
7955       else
7956         return none
7957
7958   private partial def isQuotRecStuck? (recVal : QuotVal) (recLvls : List Level) (recArgs : Array Expr) : MetaM (Option MVarId) :=
7959     let process? (majorPos : Nat) : MetaM (Option MVarId) :=
7960       if h : majorPos < recArgs.size then do
7961         let major := recArgs.get (majorPos, h)
7962         let major ← whnf major
7963         getStuckMVar? major
7964       else
7965         return none
7966     match recVal.kind with
7967     | QuotKind.lift => process? 5
7968     | QuotKind.ind  => process? 4
7969     | _             => return none
7970
7971   /-- Return `some (Expr.mvar mvarId)` if metavariable `mvarId` is blocking reduction. -/
7972   partial def getStuckMVar? : Expr → MetaM (Option MVarId)
7973   | Expr.mdata _ e _      => getStuckMVar? e
7974   | Expr.proj _ _ e _    => do getStuckMVar? (← whnf e)
7975   | e@(Expr.mvar ..) => do
7976     let e ← instantiateMVars e
7977     match e with
7978     | Expr.mvar mvarId _ => pure (some mvarId)
7979     | _ => getStuckMVar? e
7980   | e@(Expr.app f _ _) =>
7981     let f := f.getAppFn
7982     match f with
7983     | Expr.mvar mvarId _      => return some mvarId
7984     | Expr.const fName fLvls _ => do
7985       let cinfo? ← getConstNoEx? fName
7986       match cinfo? with
7987       | some $ ConstantInfo.recInfo recVal => isRecStuck? recVal fLvls e.getAppArgs
7988       | some $ ConstantInfo.quotInfo recVal => isQuotRecStuck? recVal fLvls e.getAppArgs
7989       | _ => return none
7990   | _ => return none

```

```

7991   | _ => return none
7992 end
7993
7994 /- =====
7995   Weak Head Normal Form auxiliary combinators
7996   ===== -/
7997
7998 /- Auxiliary combinator for handling easy WHNF cases. It takes a function for handling the "hard" cases as an argument -/
7999 @[specialize] private partial def whnfEasyCases (e : Expr) (k : Expr → MetaM Expr) : MetaM Expr := do
8000   match e with
8001   | Expr.forallE ..      => return e
8002   | Expr.lam ..         => return e
8003   | Expr.sort ..       => return e
8004   | Expr.lit ..        => return e
8005   | Expr.bvar ..       => unreachable!
8006   | Expr.letE ..       => k e
8007   | Expr.const ..      => k e
8008   | Expr.app ..        => k e
8009   | Expr.proj ..       => k e
8010   | Expr.mdata _ e _   => whnfEasyCases e k
8011   | Expr.fvar fvarId _ =>
8012     let decl ← getLocalDecl fvarId
8013     match decl with
8014     | LocalDecl.cdecl .. => return e
8015     | LocalDecl.ldecl (value := v) (nonDep := nonDep) .. =>
8016       let cfg ← getConfig
8017       if nonDep && !cfg.zetaNonDep then
8018         return e
8019       else
8020         when cfg.trackZeta do
8021           modify fun s => { s with zetaFVarIds := s.zetaFVarIds.insert fvarId }
8022         whnfEasyCases v k
8023   | Expr.mvar mvarId _ =>
8024     match (← getExprMVarAssignment? mvarId) with
8025     | some v => whnfEasyCases v k
8026     | none   => return e
8027
8028 /- Return true iff term is of the form `idRhs ...` -/
8029 private def isIdRhsApp (e : Expr) : Bool :=
8030   e.isAppOf `idRhs
8031
8032 /- (@idRhs T f a_1 ... a_n) ==> (f a_1 ... a_n) -/
8033 private def extractIdRhs (e : Expr) : Expr :=
8034   if !isIdRhsApp e then e
8035   else
8036     let args := e.getAppArgs
8037     if args.size < 2 then e

```

```

8038     else mkAppRange args[1] 2 args.size args
8039
8040 @[specialize] private def deltaDefinition {α} (c : ConstantInfo) (lvls : List Level)
8041   (failK : Unit → α) (successK : Expr → α) : α :=
8042   if c.levelParams.length != lvls.length then failK ()
8043   else
8044     let val := c.instantiateValueLevelParams lvls
8045     successK (extractIdRhs val)
8046
8047 @[specialize] private def deltaBetaDefinition {α} (c : ConstantInfo) (lvls : List Level) (revArgs : Array Expr)
8048   (failK : Unit → α) (successK : Expr → α) : α :=
8049   if c.levelParams.length != lvls.length then
8050     failK ()
8051   else
8052     let val := c.instantiateValueLevelParams lvls
8053     let val := val.betaRev revArgs
8054     successK (extractIdRhs val)
8055
8056 inductive ReduceMatcherResult where
8057   | reduced (val : Expr)
8058   | stuck   (val : Expr)
8059   | notMatcher
8060   | partialApp
8061
8062 def reduceMatcher? (e : Expr) : MetaM ReduceMatcherResult := do
8063   match e.getAppFn with
8064   | Expr.const declName declLevels _ =>
8065     let some info ← getMatcherInfo? declName
8066     | return ReduceMatcherResult.notMatcher
8067   let args := e.getAppArgs
8068   let prefixSz := info.numParams + 1 + info.numDiscrs
8069   if args.size < prefixSz + info.numAlts then
8070     return ReduceMatcherResult.partialApp
8071   else
8072     let constInfo ← getConstInfo declName
8073     let f := constInfo.instantiateValueLevelParams declLevels
8074     let auxApp := mkAppN f args[0:prefixSz]
8075     let auxAppType ← inferType auxApp
8076     forallBoundedTelescope auxAppType info.numAlts fun hs _ => do
8077       let auxApp := mkAppN auxApp hs
8078       let auxApp ← whnf auxApp
8079       let auxAppFn := auxApp.getAppFn
8080       let mut i := prefixSz
8081       for h in hs do
8082         if auxAppFn == h then
8083           let result := mkAppN args[i] auxApp.getAppArgs
8084           let result := mkAppN result args[prefixSz + info.numAlts:args.size]

```

```

8085         return ReduceMatcherResult.reduced result.headBeta
8086         i := i + 1
8087         return ReduceMatcherResult.stuck auxApp
8088     | _ => pure ReduceMatcherResult.notMatcher
8089
8090 /- Given an expression `e`, compute its WHNF and if the result is a constructor, return field #i. -/
8091 def project? (e : Expr) (i : Nat) : MetaM (Option Expr) := do
8092   let e ← whnf e
8093   matchConstCtor e.getAppFn (fun _ => pure none) fun ctorVal _ =>
8094     let numArgs := e.getAppNumArgs
8095     let idx := ctorVal.numParams + i
8096     if idx < numArgs then
8097       return some (e.getArg! idx)
8098     else
8099       return none
8100
8101 def reduceProj? (e : Expr) : MetaM (Option Expr) := do
8102   match e with
8103   | Expr.proj _ i c _ => project? c i
8104   | _                 => return none
8105
8106 /-
8107   Auxiliary method for reducing terms of the form `?m t_1 ... t_n` where `?m` is delayed assigned.
8108   Recall that we can only expand a delayed assignment when all holes/metavariables in the assigned value have been "filled".
8109 -/
8110 private def whnfDelayedAssigned? (f' : Expr) (e : Expr) : MetaM (Option Expr) := do
8111   if f'.isMVar then
8112     match (← getDelayedAssignment? f'.mvarId!) with
8113     | none => return none
8114     | some { fvars := fvars, val := val, .. } =>
8115       let args := e.getAppArgs
8116       if fvars.size > args.size then
8117         -- Insufficient number of argument to expand delayed assignment
8118         return none
8119       else
8120         let newVal ← instantiateMVars val
8121         if newVal.hasExprMVar then
8122           -- Delayed assignment still contains metavariables
8123           return none
8124         else
8125           let newVal := newVal.abstract fvars
8126           let result := newVal.instantiateRevRange 0 fvars.size args
8127           return mkAppRange result fvars.size args.size args
8128   else
8129     return none
8130
8131 /--

```

```

8132 Apply beta-reduction, zeta-reduction (i.e., unfold let local-decls), iota-reduction,
8133 expand let-expressions, expand assigned meta-variables. -/
8134 partial def whnfCore (e : Expr) : MetaM Expr :=
8135   whnfEasyCases e fun e => do
8136     trace[Meta.whnf]! e
8137     match e with
8138     | Expr.const .. => pure e
8139     | Expr.letE _ _ v b _ => whnfCore $ b.instantiate1 v
8140     | Expr.app f .. =>
8141       let f := f.getAppFn
8142       let f' ← whnfCore f
8143       if f'.isLambda then
8144         let revArgs := e.getAppRevArgs
8145         whnfCore <| f'.betaRev revArgs
8146       else if let some eNew ← whnfDelayedAssigned? f' e then
8147         whnfCore eNew
8148       else
8149         let e := if f == f' then e else e.updateFn f'
8150         match (← reduceMatcher? e) with
8151         | ReduceMatcherResult.reduced eNew => whnfCore eNew
8152         | ReduceMatcherResult.partialApp   => pure e
8153         | ReduceMatcherResult.stuck _      => pure e
8154         | ReduceMatcherResult.notMatcher   =>
8155           matchConstAux f' (fun _ => return e) fun cinfo lvls =>
8156             match cinfo with
8157             | ConstantInfo.recInfo rec    => reduceRec rec lvls e.getAppArgs (fun _ => return e) whnfCore
8158             | ConstantInfo.quotInfo rec   => reduceQuotRec rec lvls e.getAppArgs (fun _ => return e) whnfCore
8159             | c@(ConstantInfo.defnInfo _) => do
8160               if (← isAuxDef c.name) then
8161                 deltaBetaDefinition c lvls e.getAppRevArgs (fun _ => return e) whnfCore
8162               else
8163                 return e
8164             | _ => return e
8165     | Expr.proj .. => match (← reduceProj? e) with
8166     | some e => whnfCore e
8167     | none => return e
8168     | _ => unreachable!
8169
8170 mutual
8171 /-- Reduce `e` until `idRhs` application is exposed or it gets stuck.
8172 This is a helper method for implementing smart unfolding. -/
8173 private partial def whnfUntilIdRhs (e : Expr) : MetaM Expr := do
8174   let e ← whnfCore e
8175   match (← getStuckMVar? e) with
8176   | some mvarId =>
8177     /- Try to "unstuck" by resolving pending TC problems -/
8178     if (← Meta.synthPending mvarId) then

```

```

8179     whnfUntilIdRhs e
8180   else
8181     return e -- failed because metavariable is blocking reduction
8182 | _ =>
8183   if isIdRhsApp e then
8184     return e -- done
8185   else
8186     match (← unfoldDefinition? e) with
8187     | some e => whnfUntilIdRhs e
8188     | none   => pure e -- failed because of symbolic argument
8189
8190 /--
8191   Auxiliary method for unfolding a class projection when transparency is set to `TransparencyMode.instances`.
8192   Recall that that class instance projections are not marked with `[reducible]` because we want them to be
8193   in "reducible canonical form".
8194 -/
8195 private partial def unfoldProjInst (e : Expr) : MetaM (Option Expr) := do
8196   if (← getTransparency) != TransparencyMode.instances then
8197     return none
8198   else
8199     match e.getAppFn with
8200     | Expr.const declName .. =>
8201       match (← getProjectionFnInfo? declName) with
8202       | some { fromClass := true, .. } =>
8203         match (← withDefault <| unfoldDefinition? e) with
8204         | none   => return none
8205         | some e =>
8206           match (← reduceProj? e.getAppFn) with
8207           | none   => return none
8208           | some r => return mkAppN r e.getAppArgs |>.headBeta
8209     | _ => return none
8210 | _ => return none
8211
8212 /-- Unfold definition using "smart unfolding" if possible. -/
8213 partial def unfoldDefinition? (e : Expr) : MetaM (Option Expr) :=
8214   match e with
8215   | Expr.app f _ _ =>
8216     matchConstAux f.getAppFn (fun _ => unfoldProjInst e) fun fInfo fLvls => do
8217       if fInfo.levelParams.length != fLvls.length then
8218         return none
8219     else
8220       let unfoldDefault (_ : Unit) : MetaM (Option Expr) :=
8221         if fInfo.hasValue then
8222           deltaBetaDefinition fInfo fLvls e.getAppRevArgs (fun _ => pure none) (fun e => pure (some e))
8223         else
8224           return none
8225       if smartUnfolding.get (← getOptions) then

```



```

8226         let fAuxInfo? ← getConstNoEx? (mkSmartUnfoldingNameFor fInfo.name)
8227         match fAuxInfo? with
8228         | some fAuxInfo@(ConstantInfo.defnInfo _) =>
8229             deltaBetaDefinition fAuxInfo fLvls e.getAppRevArgs (fun _ => pure none) fun e₁ => do
8230                 let e₂ ← whnfUntilIdRhs e₁
8231                 if isIdRhsApp e₂ then
8232                     return some (extractIdRhs e₂)
8233                 else
8234                     return none
8235         | _ => unfoldDefault ()
8236     else
8237         unfoldDefault ()
8238 | Expr.const declName lvls _ => do
8239     if smartUnfolding.get (← getOptions) && (← getEnv).contains (mkSmartUnfoldingNameFor declName) then
8240         return none
8241     else
8242         let (some (cinfo@(ConstantInfo.defnInfo _))) ← getConstNoEx? declName | pure none
8243         deltaDefinition cinfo lvls
8244             (fun _ => pure none)
8245             (fun e => pure (some e))
8246 | _ => return none
8247 end
8248
8249 def unfoldDefinition (e : Expr) : MetaM Expr := do
8250     let some e ← unfoldDefinition? e | throwError! "failed to unfold definition{indentExpr e}"
8251     return e
8252
8253 @[specialize] partial def whnfHeadPred (e : Expr) (pred : Expr → MetaM Bool) : MetaM Expr :=
8254     whnfEasyCases e fun e => do
8255         let e ← whnfCore e
8256         if (← pred e) then
8257             match (← unfoldDefinition? e) with
8258             | some e => whnfHeadPred e pred
8259             | none    => return e
8260         else
8261             return e
8262
8263 def whnfUntil (e : Expr) (declName : Name) : MetaM (Option Expr) := do
8264     let e ← whnfHeadPred e (fun e => return !e.isAppOf declName)
8265     if e.isAppOf declName then
8266         return e
8267     else
8268         return none
8269
8270 /-- Try to reduce matcher/recursor/quot applications. We say they are all "morally" recursor applications. -/
8271 def reduceRecMatcher? (e : Expr) : MetaM (Option Expr) := do
8272     if !e.isApp then

```

```

8273     return none
8274   else match (← reduceMatcher? e) with
8275   | ReduceMatcherResult.reduced e => return e
8276   | _ => matchConstAux e.getAppFn (fun _ => pure none) fun cinfo lvls => do
8277     match cinfo with
8278     | ConstantInfo.recInfo «rec» => reduceRec «rec» lvls e.getAppArgs (fun _ => pure none) (fun e => pure (some e))
8279     | ConstantInfo.quotInfo «rec» => reduceQuotRec «rec» lvls e.getAppArgs (fun _ => pure none) (fun e => pure (some e))
8280     | c@(ConstantInfo.defnInfo _) =>
8281       if (← isAuxDef c.name) then
8282         deltaBetaDefinition c lvls e.getAppRevArgs (fun _ => pure none) (fun e => pure (some e))
8283       else
8284         return none
8285     | _ => return none
8286
8287 unsafe def reduceBoolNativeUnsafe (constName : Name) : MetaM Bool := evalConstCheck Bool `Bool constName
8288 unsafe def reduceNatNativeUnsafe (constName : Name) : MetaM Nat := evalConstCheck Nat `Nat constName
8289 @[implementedBy reduceBoolNativeUnsafe] constant reduceBoolNative (constName : Name) : MetaM Bool
8290 @[implementedBy reduceNatNativeUnsafe] constant reduceNatNative (constName : Name) : MetaM Nat
8291
8292 def reduceNative? (e : Expr) : MetaM (Option Expr) :=
8293   match e with
8294   | Expr.app (Expr.const fName _ _) (Expr.const argName _ _) _ =>
8295     if fName == `Lean.reduceBool then do
8296       return toExpr (← reduceBoolNative argName)
8297     else if fName == `Lean.reduceNat then do
8298       return toExpr (← reduceNatNative argName)
8299     else
8300       return none
8301   | _ =>
8302     return none
8303
8304 @[inline] def withNatValue {α} (a : Expr) (k : Nat → MetaM (Option α)) : MetaM (Option α) := do
8305   let a ← whnf a
8306   match a with
8307   | Expr.const `Nat.zero _ _ => k 0
8308   | Expr.lit (Literal.natVal v) _ => k v
8309   | _ => return none
8310
8311 def reduceUnaryNatOp (f : Nat → Nat) (a : Expr) : MetaM (Option Expr) :=
8312   withNatValue a fun a =>
8313     return mkNatLit <| f a
8314
8315 def reduceBinNatOp (f : Nat → Nat → Nat) (a b : Expr) : MetaM (Option Expr) :=
8316   withNatValue a fun a =>
8317     withNatValue b fun b => do
8318       trace[Meta.isDefEq.whnf.reduceBinOp]! "{a} op {b}"
8319       return mkNatLit <| f a b

```

```

8320
8321 def reduceBinNatPred (f : Nat → Nat → Bool) (a b : Expr) : MetaM (Option Expr) := do
8322   withNatValue a fun a =>
8323     withNatValue b fun b =>
8324       return toExpr <| f a b
8325
8326 def reduceNat? (e : Expr) : MetaM (Option Expr) :=
8327   if e.hasFVar || e.hasMVar then
8328     return none
8329   else match e with
8330     | Expr.app (Expr.const fn _ _) a _ =>
8331       if fn == `Nat.succ then
8332         reduceUnaryNatOp Nat.succ a
8333       else
8334         return none
8335     | Expr.app (Expr.app (Expr.const fn _ _) a1 _) a2 _ =>
8336       if fn == `Nat.add then reduceBinNatOp Nat.add a1 a2
8337       else if fn == `Nat.sub then reduceBinNatOp Nat.sub a1 a2
8338       else if fn == `Nat.mul then reduceBinNatOp Nat.mul a1 a2
8339       else if fn == `Nat.div then reduceBinNatOp Nat.div a1 a2
8340       else if fn == `Nat.mod then reduceBinNatOp Nat.mod a1 a2
8341       else if fn == `Nat.beq then reduceBinNatPred Nat.beq a1 a2
8342       else if fn == `Nat.ble then reduceBinNatPred Nat.ble a1 a2
8343       else return none
8344     | _ =>
8345       return none
8346
8347
8348 @[inline] private def useWHNFCache (e : Expr) : MetaM Bool := do
8349   -- We cache only closed terms without expr metavar.
8350   -- Potential refinement: cache if `e` is not stuck at a metavariable
8351   if e.hasFVar || e.hasExprMVar then
8352     return false
8353   else
8354     match (← getConfig).transparency with
8355     | TransparencyMode.default => true
8356     | TransparencyMode.all     => true
8357     | _                        => false
8358
8359 @[inline] private def cached? (useCache : Bool) (e : Expr) : MetaM (Option Expr) := do
8360   if useCache then
8361     match (← getConfig).transparency with
8362     | TransparencyMode.default => return (← get).cache.whnfDefault.find? e
8363     | TransparencyMode.all     => return (← get).cache.whnfAll.find? e
8364     | _                        => unreachable!
8365   else
8366     return none

```

```

8367
8368 private def cache (useCache : Bool) (e r : Expr) : MetaM Expr := do
8369   if useCache then
8370     match (← getConfig).transparency with
8371     | TransparencyMode.default => modify fun s => { s with cache.whnfDefault := s.cache.whnfDefault.insert e r }
8372     | TransparencyMode.all     => modify fun s => { s with cache.whnfAll      := s.cache.whnfAll.insert e r }
8373     | _                       => unreachable!
8374   return r
8375
8376 partial def whnfImp (e : Expr) : MetaM Expr :=
8377   whnfEasyCases e fun e => do
8378     checkMaxHeartbeats "whnf"
8379     let useCache ← useWHNFCache e
8380     match (← cached? useCache e) with
8381     | some e' => pure e'
8382     | none   =>
8383       let e' ← whnfCore e
8384       match (← reduceNat? e') with
8385       | some v => cache useCache e v
8386       | none   =>
8387         match (← reduceNative? e') with
8388         | some v => cache useCache e v
8389         | none   =>
8390           match (← unfoldDefinition? e') with
8391           | some e => whnfImp e
8392           | none   => cache useCache e e'
8393
8394 @[builtinInit] def setWHNFRef : IO Unit :=
8395   whnfRef.set whnfImp
8396
8397 builtin_initialize
8398   registerTraceClass `Meta.whnf
8399
8400 end Lean.Meta

```