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

namespace Lean.Meta

structure AbstractMVarsResult where
  paramNames : Array Name
  numMVars   : Nat
  expr       : Expr
  deriving Inhabited, BEq

namespace AbstractMVars

open Std (HashMap)

structure State where
  ngen         : NameGenerator
  lctx         : LocalContext
  nextParamIdx : Nat := 0
  paramNames   : Array Name := #[]
  fvars        : Array Expr  := #[]
  lmap         : HashMap Name Level := {}
  emap         : HashMap Name Expr  := {}

abbrev M := ReaderT MetavarContext (StateM State)

def mkFreshId : M Name := do
  let s ← get
  let fresh := s.ngen.curr
  modify fun s => { s with ngen := s.ngen.next }
  pure fresh

private partial def abstractLevelMVars (u : Level) : M Level := do
  if !u.hasMVar then
    return u
  else
    match u with
    | Level.zero _        => return u
    | Level.param _ _     => return u
```

```
    | Level.succ v _      => return u.updateSucc! (← abstractLevelMVars v)
    | Level.max v w _     => return u.updateMax! (← abstractLevelMVars v) (← abstractLevelMVars w)
    | Level.imax v w _    => return u.updateIMax! (← abstractLevelMVars v) (← abstractLevelMVars w)
    | Level.mvar mvarId _ =>
      let mctx ← read
      let depth := mctx.getLevelDepth mvarId;
      if depth != mctx.depth then
        return u -- metavariables from lower depths are treated as constants
      else
        let s ← get
        match s.lmap.find? mvarId with
        | some u => pure u
        | none   =>
          let paramId := Name.mkNum `_abstMVar s.nextParamIdx
          let u := mkLevelParam paramId
          modify fun s => { s with nextParamIdx := s.nextParamIdx + 1, lmap := s.lmap.insert mvarId u, paramNames := s.paramNames.push paramId }
          return u

partial def abstractExprMVars (e : Expr) : M Expr := do
  if !e.hasMVar then
    return e
  else
    match e with
    | e@(Expr.lit _ _)         => return e
    | e@(Expr.bvar _ _)        => return e
    | e@(Expr.fvar _ _)        => return e
    | e@(Expr.sort u _)        => return e.updateSort! (← abstractLevelMVars u)
    | e@(Expr.const _ us _)    => return e.updateConst! (← us.mapM abstractLevelMVars)
    | e@(Expr.proj _ _ s _)    => return e.updateProj! (← abstractExprMVars s)
    | e@(Expr.app f a _)       => return e.updateApp! (← abstractExprMVars f) (← abstractExprMVars a)
    | e@(Expr.mdata _ b _)     => return e.updateMData! (← abstractExprMVars b)
    | e@(Expr.lam _ d b _)     => return e.updateLambdaE! (← abstractExprMVars d) (← abstractExprMVars b)
    | e@(Expr.forallE _ d b _) => return e.updateForallE! (← abstractExprMVars d) (← abstractExprMVars b)
    | e@(Expr.letE _ t v b _)  => return e.updateLet! (← abstractExprMVars t) (← abstractExprMVars v) (← abstractExprMVars b)
    | e@(Expr.mvar mvarId _)   =>
      let mctx ← read
      let decl := mctx.getDecl mvarId
      if decl.depth != mctx.depth then
        return e
      else
        let s ← get
        match s.emap.find? mvarId with
        | some e =>
          return e
        | none   =>
          let type   ← abstractExprMVars decl.type
```

```
          let fvarId ← mkFreshId
          let fvar := mkFVar fvarId;
          let userName := if decl.userName.isAnonymous then (`x).appendIndexAfter s.fvars.size else decl.userName
          modify fun s => {
            s with
            emap  := s.emap.insert mvarId fvar,
            fvars := s.fvars.push fvar,
            lctx  := s.lctx.mkLocalDecl fvarId userName type }
          return fvar

end AbstractMVars

/--
  Abstract (current depth) metavariables occurring in `e`.
  The result contains
  - An array of universe level parameters that replaced universe metavariables occurring in `e`.
  - The number of (expr) metavariables abstracted.
  - And an expression of the form `fun (m_1 : A_1) ... (m_k : A_k) => e'`, where
    `k` equal to the number of (expr) metavariables abstracted, and `e'` is `e` after we
    replace the metavariables.

  Example: given `f.{?u} ?m1` where `?m1 : ?m2 Nat`, `?m2 : Type -> Type`. This function returns
  `{ levels := #[u], size := 2, expr := (fun (m2 : Type -> Type) (m1 : m2 Nat) => f.{u} m1) }`

  This API can be used to "transport" to a different metavariable context.
  Given a new metavariable context, we replace the `AbstractMVarsResult.levels` with
  new fresh universe metavariables, and instantiate the `(m_i : A_i)` in the lambda-expression
  with new fresh metavariables.

  Application: we use this method to cache the results of type class resolution. -/
def abstractMVars (e : Expr) : MetaM AbstractMVarsResult := do
  let e ← instantiateMVars e
  let (e, s) := AbstractMVars.abstractExprMVars e (← getMCtx) { lctx := (← getLCtx), ngen := (← getNGen) }
  setNGen s.ngen
  let e := s.lctx.mkLambda s.fvars e
  pure { paramNames := s.paramNames, numMVars := s.fvars.size, expr := e }

def openAbstractMVarsResult (a : AbstractMVarsResult) : MetaM (Array Expr × Array BinderInfo × Expr) := do
  let us ← a.paramNames.mapM fun _ => mkFreshLevelMVar
  let e := a.expr.instantiateLevelParamsArray a.paramNames us
  lambdaMetaTelescope e (some a.numMVars)

end Lean.Meta
// ::::::::::::::
// AbstractNestedProofs.lean
// ::::::::::::::
/-
```

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

namespace Lean.Meta
namespace AbstractNestedProofs

def isNonTrivialProof (e : Expr) : MetaM Bool := do
  if !(← isProof e) then
    pure false
  else
    e.withApp fun f args =>
      pure $ !f.isAtomic || args.any fun arg => !arg.isAtomic

structure Context where
  baseName : Name

structure State where
  nextIdx : Nat := 1

abbrev M := ReaderT Context $ MonadCacheT Expr Expr $ StateRefT State MetaM

private def mkAuxLemma (e : Expr) : M Expr := do
  let ctx ← read
  let s ← get
  let lemmaName ← mkAuxName (ctx.baseName ++ `proof) s.nextIdx
  modify fun s => { s with nextIdx := s.nextIdx + 1 }
  mkAuxDefinitionFor lemmaName e

partial def visit (e : Expr) : M Expr := do
  if e.isAtomic then
    pure e
  else
    let visitBinders (xs : Array Expr) (k : M Expr) : M Expr := do
      let localInstances ← getLocalInstances
      let mut lctx ← getLCtx
      for x in xs do
        let xFVarId := x.fvarId!
        let localDecl ← getLocalDecl xFVarId
        let type      ← visit localDecl.type
        let localDecl := localDecl.setType type
        let localDecl ← match localDecl.value? with
           | some value => do let value ← visit value; pure $ localDecl.setValue value
           | none       => pure localDecl
        lctx :=lctx.modifyLocalDecl xFVarId fun _ => localDecl
```

```
      withLCtx lctx localInstances k
    checkCache e fun _ => do
      if (← isNonTrivialProof e) then
        mkAuxLemma e
      else match e with
        | Expr.lam _ _ _ _     => lambdaLetTelescope e fun xs b => visitBinders xs do mkLambdaFVars xs (← visit b)
        | Expr.letE _ _ _ _ _  => lambdaLetTelescope e fun xs b => visitBinders xs do mkLambdaFVars xs (← visit b)
        | Expr.forallE _ _ _ _ => forallTelescope e fun xs b => visitBinders xs do mkForallFVars xs (← visit b)
        | Expr.mdata _ b _     => return e.updateMData! (← visit b)
        | Expr.proj _ _ b _    => return e.updateProj! (← visit b)
        | Expr.app _ _ _       => e.withApp fun f args => return mkAppN f (← args.mapM visit)
        | _                    => pure e

end AbstractNestedProofs

/-- Replace proofs nested in `e` with new lemmas. The new lemmas have names of the form `mainDeclName.proof_<idx>` -/
def abstractNestedProofs (mainDeclName : Name) (e : Expr) : MetaM Expr :=
  AbstractNestedProofs.visit e |>.run { baseName := mainDeclName } |>.run |>.run' { nextIdx := 1 }

end Lean.Meta
// ::::::::::::::
// AppBuilder.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Structure
import Lean.Util.Recognizers
import Lean.Meta.SynthInstance
import Lean.Meta.Check

namespace Lean.Meta

/-- Return `id e` -/
def mkId (e : Expr) : MetaM Expr := do
  let type ← inferType e
  let u    ← getLevel type
  return mkApp2 (mkConst ``id [u]) type e

/-- Return `idRhs e` -/
def mkIdRhs (e : Expr) : MetaM Expr :=  do
  let type ← inferType e
  let u    ← getLevel type
  return mkApp2 (mkConst ``idRhs [u]) type e

```

```
/--
  Given `e` s.t. `inferType e` is definitionally equal to `expectedType`, return
  term `@id expectedType e`. -/
def mkExpectedTypeHint (e : Expr) (expectedType : Expr) : MetaM Expr := do
  let u ← getLevel expectedType
  return mkApp2 (mkConst ``id [u]) expectedType e

def mkEq (a b : Expr) : MetaM Expr := do
  let aType ← inferType a
  let u ← getLevel aType
  return mkApp3 (mkConst ``Eq [u]) aType a b

def mkHEq (a b : Expr) : MetaM Expr := do
  let aType ← inferType a
  let bType ← inferType b
  let u ← getLevel aType
  return mkApp4 (mkConst ``HEq [u]) aType a bType b

def mkEqRefl (a : Expr) : MetaM Expr := do
  let aType ← inferType a
  let u ← getLevel aType
  return mkApp2 (mkConst ``Eq.refl [u]) aType a

def mkHEqRefl (a : Expr) : MetaM Expr := do
  let aType ← inferType a
  let u ← getLevel aType
  return mkApp2 (mkConst ``HEq.refl [u]) aType a

def mkAbsurd (e : Expr) (hp hnp : Expr) : MetaM Expr := do
  let p ← inferType hp
  let u ← getLevel e
  return mkApp4 (mkConst ``absurd [u]) p e hp hnp

def mkFalseElim (e : Expr) (h : Expr) : MetaM Expr := do
  let u ← getLevel e
  return mkApp2 (mkConst ``False.elim [u]) e h

private def infer (h : Expr) : MetaM Expr := do
  let hType ← inferType h
  whnfD hType

private def hasTypeMsg (e type : Expr) : MessageData :=
  m!"{indentExpr e}\nhas type{indentExpr type}"

private def throwAppBuilderException {α} (op : Name) (msg : MessageData) : MetaM α :=
  throwError! "AppBuilder for '{op}', {msg}"

```

```
def mkEqSymm (h : Expr) : MetaM Expr := do
  if h.isAppOf ``Eq.refl then
    return h
  else
    let hType ← infer h
    match hType.eq? with
    | some (α, a, b) =>
      let u ← getLevel α
      return mkApp4 (mkConst ``Eq.symm [u]) α a b h
    | none => throwAppBuilderException ``Eq.symm ("equality proof expected" ++ hasTypeMsg h hType)

def mkEqTrans (h₁ h₂ : Expr) : MetaM Expr := do
  if h₁.isAppOf ``Eq.refl then
    return h₂
  else if h₂.isAppOf ``Eq.refl then
    return h₁
  else
    let hType₁ ← infer h₁
    let hType₂ ← infer h₂
    match hType₁.eq?, hType₂.eq? with
    | some (α, a, b), some (_, _, c) =>
      let u ← getLevel α
      return mkApp6 (mkConst ``Eq.trans [u]) α a b c h₁ h₂
    | none, _ => throwAppBuilderException ``Eq.trans ("equality proof expected" ++ hasTypeMsg h₁ hType₁)
    | _, none => throwAppBuilderException ``Eq.trans ("equality proof expected" ++ hasTypeMsg h₂ hType₂)

def mkHEqSymm (h : Expr) : MetaM Expr := do
  if h.isAppOf ``HEq.refl then
    return h
  else
    let hType ← infer h
    match hType.heq? with
    | some (α, a, β, b) =>
      let u ← getLevel α
      return mkApp5 (mkConst ``HEq.symm [u]) α β a b h
    | none =>
      throwAppBuilderException ``HEq.symm ("heterogeneous equality proof expected" ++ hasTypeMsg h hType)

def mkHEqTrans (h₁ h₂ : Expr) : MetaM Expr := do
  if h₁.isAppOf ``HEq.refl then
    return h₂
  else if h₂.isAppOf ``HEq.refl then
    return h₁
  else
    let hType₁ ← infer h₁
    let hType₂ ← infer h₂
    match hType₁.heq?, hType₂.heq? with
```

```
    | some (α, a, β, b), some (_, _, γ, c) =>
      let u ← getLevel α
      return mkApp8 (mkConst ``HEq.trans [u]) α β γ a b c h₁ h₂
    | none, _ => throwAppBuilderException ``HEq.trans ("heterogeneous equality proof expected" ++ hasTypeMsg h₁ hType₁)
    | _, none => throwAppBuilderException ``HEq.trans ("heterogeneous equality proof expected" ++ hasTypeMsg h₂ hType₂)

def mkEqOfHEq (h : Expr) : MetaM Expr := do
  let hType ← infer h
  match hType.heq? with
  | some (α, a, β, b) =>
    unless (← isDefEq α β) do
      throwAppBuilderException ``eqOfHEq m!"heterogeneous equality types are not definitionally equal{indentExpr α}\nis not definitionally equal to{indentExpr β}"
    let u ← getLevel α
    return mkApp4 (mkConst ``eqOfHEq [u]) α a b h
  | _ =>
    throwAppBuilderException ``HEq.trans m!"heterogeneous equality proof expected{indentExpr h}"

def mkCongrArg (f h : Expr) : MetaM Expr := do
  if h.isAppOf ``Eq.refl then
    mkEqRefl (mkApp f h.appArg!)
  else
    let hType ← infer h
    let fType ← infer f
    match fType.arrow?, hType.eq? with
    | some (α, β), some (_, a, b) =>
      let u ← getLevel α
      let v ← getLevel β
      return mkApp6 (mkConst ``congrArg [u, v]) α β a b f h
    | none, _ => throwAppBuilderException ``congrArg ("non-dependent function expected" ++ hasTypeMsg f fType)
    | _, none => throwAppBuilderException ``congrArg ("equality proof expected" ++ hasTypeMsg h hType)

def mkCongrFun (h a : Expr) : MetaM Expr := do
  if h.isAppOf ``Eq.refl then
    mkEqRefl (mkApp h.appArg! a)
  else
    let hType ← infer h
    match hType.eq? with
    | some (ρ, f, g) => do
      let ρ ← whnfD ρ
      match ρ with
      | Expr.forallE n α β _ =>
        let β' := Lean.mkLambda n BinderInfo.default α β
        let u ← getLevel α
        let v ← getLevel (mkApp β' a)
        return mkApp6 (mkConst ``congrFun [u, v]) α β' f g h a
      | _ => throwAppBuilderException ``congrFun ("equality proof between functions expected" ++ hasTypeMsg h hType)
```

```
    | _ => throwAppBuilderException ``congrFun ("equality proof expected" ++ hasTypeMsg h hType)

def mkCongr (h₁ h₂ : Expr) : MetaM Expr := do
  if h₁.isAppOf ``Eq.refl then
    mkCongrArg h₁.appArg! h₂
  else if h₂.isAppOf ``Eq.refl then
    mkCongrFun h₁ h₂.appArg!
  else
    let hType₁ ← infer h₁
    let hType₂ ← infer h₂
    match hType₁.eq?, hType₂.eq? with
    | some (ρ, f, g), some (α, a, b) =>
      let ρ ← whnfD ρ
      match ρ.arrow? with
      | some (_, β) => do
        let u ← getLevel α
        let v ← getLevel β
        return mkApp8 (mkConst ``congr [u, v]) α β f g a b h₁ h₂
      | _ => throwAppBuilderException ``congr ("non-dependent function expected" ++ hasTypeMsg h₁ hType₁)
    | none, _ => throwAppBuilderException ``congr ("equality proof expected" ++ hasTypeMsg h₁ hType₁)
    | _, none => throwAppBuilderException ``congr ("equality proof expected" ++ hasTypeMsg h₂ hType₂)

private def mkAppMFinal (methodName : Name) (f : Expr) (args : Array Expr) (instMVars : Array MVarId) : MetaM Expr := do
  instMVars.forM fun mvarId => do
    let mvarDecl ← getMVarDecl mvarId
    let mvarVal  ← synthInstance mvarDecl.type
    assignExprMVar mvarId mvarVal
  let result ← instantiateMVars (mkAppN f args)
  if (← hasAssignableMVar result) then throwAppBuilderException methodName ("result contains metavariables" ++ indentExpr result)
  return result

private partial def mkAppMArgs (f : Expr) (fType : Expr) (xs : Array Expr) : MetaM Expr :=
  let rec loop (type : Expr) (i : Nat) (j : Nat) (args : Array Expr) (instMVars : Array MVarId) : MetaM Expr := do
    if i >= xs.size then
      mkAppMFinal `mkAppM f args instMVars
    else match type with
      | Expr.forallE n d b c =>
        let d  := d.instantiateRevRange j args.size args
        match c.binderInfo with
        | BinderInfo.implicit     =>
          let mvar ← mkFreshExprMVar d MetavarKind.natural n
          loop b i j (args.push mvar) instMVars
        | BinderInfo.instImplicit =>
          let mvar ← mkFreshExprMVar d MetavarKind.synthetic n
          loop b i j (args.push mvar) (instMVars.push mvar.mvarId!)
        | _ =>
          let x := xs[i]
```

```
          let xType ← inferType x
          if (← isDefEq d xType) then
            loop b (i+1) j (args.push x) instMVars
          else
            throwAppTypeMismatch (mkAppN f args) x
      | type =>
        let type := type.instantiateRevRange j args.size args
        let type ← whnfD type
        if type.isForall then
          loop type i args.size args instMVars
        else
          throwAppBuilderException `mkAppM m!"too many explicit arguments provided to{indentExpr f}\narguments{indentD xs}"
  loop fType 0 0 #[] #[]

private def mkFun (constName : Name) : MetaM (Expr × Expr) := do
  let cinfo ← getConstInfo constName
  let us ← cinfo.levelParams.mapM fun _ => mkFreshLevelMVar
  let f := mkConst constName us
  let fType := cinfo.instantiateTypeLevelParams us
  return (f, fType)

/--
  Return the application `constName xs`.
  It tries to fill the implicit arguments before the last element in `xs`.

  Remark:
  ``mkAppM `arbitrary #[α]`` returns `@arbitrary.{u} α` without synthesizing
  the implicit argument occurring after `α`.
  Given a `x : (([Decidable p] → Bool) × Nat`, ``mkAppM `Prod.fst #[x]`` returns `@Prod.fst ([Decidable p] → Bool) Nat x`
-/
def mkAppM (constName : Name) (xs : Array Expr) : MetaM Expr := do
  traceCtx `Meta.appBuilder <| withNewMCtxDepth do
    let (f, fType) ← mkFun constName
    let r ← mkAppMArgs f fType xs
    trace[Meta.appBuilder]! "constName: {constName}, xs: {xs}, result: {r}"
    return r

private partial def mkAppOptMAux (f : Expr) (xs : Array (Option Expr)) : Nat → Array Expr → Nat → Array MVarId → Expr → MetaM Expr
  | i, args, j, instMVars, Expr.forallE n d b c => do
    let d  := d.instantiateRevRange j args.size args
    if h : i < xs.size then
      match xs.get ⟨i, h⟩ with
      | none =>
        match c.binderInfo with
        | BinderInfo.instImplicit => do
          let mvar ← mkFreshExprMVar d MetavarKind.synthetic n
          mkAppOptMAux f xs (i+1) (args.push mvar) j (instMVars.push mvar.mvarId!) b
```

```
        | _                      => do
          let mvar ← mkFreshExprMVar d MetavarKind.natural n
          mkAppOptMAux f xs (i+1) (args.push mvar) j instMVars b
      | some x =>
        let xType ← inferType x
        if (← isDefEq d xType) then
          mkAppOptMAux f xs (i+1) (args.push x) j instMVars b
        else
          throwAppTypeMismatch (mkAppN f args) x
    else
      mkAppMFinal `mkAppOptM f args instMVars
  | i, args, j, instMVars, type => do
    let type := type.instantiateRevRange j args.size args
    let type ← whnfD type
    if type.isForall then
      mkAppOptMAux f xs i args args.size instMVars type
    else if i == xs.size then
      mkAppMFinal `mkAppOptM f args instMVars
    else do
      let xs : Array Expr := xs.foldl (fun r x? => match x? with | none => r | some x => r.push x) #[]
      throwAppBuilderException `mkAppOptM ("too many arguments provided to" ++ indentExpr f ++ Format.line ++ "arguments" ++ xs)

/--
  Similar to `mkAppM`, but it allows us to specify which arguments are provided explicitly using `Option` type.
  Example:
  Given `Pure.pure {m : Type u → Type v} [Pure m] {α : Type u} (a : α) : m α`,
  ```
  mkAppOptM `Pure.pure #[m, none, none, a]
  ```
  returns a `Pure.pure` application if the instance `Pure m` can be synthesized, and the universes match.
  Note that,
  ```
  mkAppM `Pure.pure #[a]
  ```
  fails because the only explicit argument `(a : α)` is not sufficient for inferring the remaining arguments,
  we would need the expected type. -/
def mkAppOptM (constName : Name) (xs : Array (Option Expr)) : MetaM Expr := do
  traceCtx `Meta.appBuilder <| withNewMCtxDepth do
    let (f, fType) ← mkFun constName
    mkAppOptMAux f xs 0 #[] 0 #[] fType

def mkEqNDRec (motive h1 h2 : Expr) : MetaM Expr := do
  if h2.isAppOf ``Eq.refl then
    return h1
  else
    let h2Type ← infer h2
    match h2Type.eq? with
```

```
    | none => throwAppBuilderException ``Eq.ndrec ("equality proof expected" ++ hasTypeMsg h2 h2Type)
    | some (α, a, b) =>
      let u2 ← getLevel α
      let motiveType ← infer motive
      match motiveType with
      | Expr.forallE _ _ (Expr.sort u1 _) _ =>
        return mkAppN (mkConst ``Eq.ndrec [u1, u2]) #[α, a, motive, h1, b, h2]
      | _ => throwAppBuilderException ``Eq.ndrec ("invalid motive" ++ indentExpr motive)

def mkEqRec (motive h1 h2 : Expr) : MetaM Expr := do
  if h2.isAppOf ``Eq.refl then
    return h1
  else
    let h2Type ← infer h2
    match h2Type.eq? with
    | none => throwAppBuilderException ``Eq.rec ("equality proof expected" ++ indentExpr h2)
    | some (α, a, b) =>
      let u2 ← getLevel α
      let motiveType ← infer motive
      match motiveType with
      | Expr.forallE _ _ (Expr.forallE _ _ (Expr.sort u1 _) _) _ =>
        return mkAppN (mkConst ``Eq.rec [u1, u2]) #[α, a, motive, h1, b, h2]
      | _ =>
        throwAppBuilderException ``Eq.rec ("invalid motive" ++ indentExpr motive)

def mkEqMP (eqProof pr : Expr) : MetaM Expr :=
  mkAppM ``Eq.mp #[eqProof, pr]

def mkEqMPR (eqProof pr : Expr) : MetaM Expr :=
  mkAppM ``Eq.mpr #[eqProof, pr]

def mkNoConfusion (target : Expr) (h : Expr) : MetaM Expr := do
  let type ← inferType h
  let type ← whnf type
  match type.eq? with
  | none           => throwAppBuilderException `noConfusion ("equality expected" ++ hasTypeMsg h type)
  | some (α, a, b) =>
    let α ← whnf α
    matchConstInduct α.getAppFn (fun _ => throwAppBuilderException `noConfusion ("inductive type expected" ++ indentExpr α)) fun v us => do
      let u ← getLevel target
      return mkAppN (mkConst (Name.mkStr v.name "noConfusion") (u :: us)) (α.getAppArgs ++ #[target, a, b, h])

def mkPure (monad : Expr) (e : Expr) : MetaM Expr :=
  mkAppOptM ``Pure.pure #[monad, none, none, e]

/--
```

```
  `mkProjection s fieldName` return an expression for accessing field `fieldName` of the structure `s`.
  Remark: `fieldName` may be a subfield of `s`. -/
partial def mkProjection : Expr → Name → MetaM Expr
  | s, fieldName => do
    let type ← inferType s
    let type ← whnf type
    match type.getAppFn with
    | Expr.const structName us _ =>
      let env ← getEnv
      unless isStructureLike env structName do
        throwAppBuilderException `mkProjection ("structure expected" ++ hasTypeMsg s type)
      match getProjFnForField? env structName fieldName with
      | some projFn =>
        let params := type.getAppArgs
        return mkApp (mkAppN (mkConst projFn us) params) s
      | none =>
        let fields := getStructureFields env structName
        let r? ← fields.findSomeM? fun fieldName' => do
          match isSubobjectField? env structName fieldName' with
          | none   => pure none
          | some _ =>
            let parent ← mkProjection s fieldName'
            (do let r ← mkProjection parent fieldName; return some r)
            <|>
            pure none
        match r? with
        | some r => pure r
        | none   => throwAppBuilderException `mkProjectionn ("invalid field name '" ++ toString fieldName ++ "' for" ++ hasTypeMsg s type)
    | _ => throwAppBuilderException `mkProjectionn ("structure expected" ++ hasTypeMsg s type)

private def mkListLitAux (nil : Expr) (cons : Expr) : List Expr → Expr
  | []    => nil
  | x::xs => mkApp (mkApp cons x) (mkListLitAux nil cons xs)

def mkListLit (type : Expr) (xs : List Expr) : MetaM Expr := do
  let u   ← getDecLevel type
  let nil := mkApp (mkConst ``List.nil [u]) type
  match xs with
  | [] => return nil
  | _  =>
    let cons := mkApp (mkConst ``List.cons [u]) type
    return mkListLitAux nil cons xs

def mkArrayLit (type : Expr) (xs : List Expr) : MetaM Expr := do
  let u ← getDecLevel type
  let listLit ← mkListLit type xs
```

```
  return mkApp (mkApp (mkConst ``List.toArray [u]) type) listLit

def mkSorry (type : Expr) (synthetic : Bool) : MetaM Expr := do
  let u ← getLevel type
  return mkApp2 (mkConst ``sorryAx [u]) type (toExpr synthetic)

/-- Return `Decidable.decide p` -/
def mkDecide (p : Expr) : MetaM Expr :=
  mkAppOptM ``Decidable.decide #[p, none]

/-- Return a proof for `p : Prop` using `decide p` -/
def mkDecideProof (p : Expr) : MetaM Expr := do
  let decP      ← mkDecide p
  let decEqTrue ← mkEq decP (mkConst ``Bool.true)
  let h         ← mkEqRefl (mkConst ``Bool.true)
  let h         ← mkExpectedTypeHint h decEqTrue
  mkAppM ``ofDecideEqTrue #[h]

/-- Return `a < b` -/
def mkLt (a b : Expr) : MetaM Expr :=
  mkAppM ``HasLess.Less #[a, b]

/-- Return `a <= b` -/
def mkLe (a b : Expr) : MetaM Expr :=
  mkAppM ``HasLessEq.LessEq #[a, b]

/-- Return `arbitrary α` -/
def mkArbitrary (α : Expr) : MetaM Expr :=
  mkAppOptM ``arbitrary #[α, none]

/-- Return `sorryAx type` -/
def mkSyntheticSorry (type : Expr) : MetaM Expr :=
  return mkApp2 (mkConst ``sorryAx [← getLevel type]) type (mkConst ``Bool.true)

/-- Return `funext h` -/
def mkFunExt (h : Expr) : MetaM Expr :=
  mkAppM ``funext #[h]

/-- Return `propext h` -/
def mkPropExt (h : Expr) : MetaM Expr :=
  mkAppM ``propext #[h]

/-- Return `ofEqTrue h` -/
def mkOfEqTrue (h : Expr) : MetaM Expr :=
  mkAppM ``ofEqTrue #[h]

/-- Return `eqTrue h` -/
```

```
def mkEqTrue (h : Expr) : MetaM Expr :=
  mkAppM ``eqTrue #[h]

/--
  Return `eqFalse h`
  `h` must have type definitionally equal to `¬ p` in the current
  reducibility setting. -/
def mkEqFalse (h : Expr) : MetaM Expr :=
  mkAppM ``eqFalse #[h]

/--
  Return `eqFalse' h`
  `h` must have type definitionally equal to `p → False` in the current
  reducibility setting. -/
def mkEqFalse' (h : Expr) : MetaM Expr :=
  mkAppM ``eqFalse' #[h]

def mkImpCongr (h₁ h₂ : Expr) : MetaM Expr :=
  mkAppM ``impCongr #[h₁, h₂]

def mkImpCongrCtx (h₁ h₂ : Expr) : MetaM Expr :=
  mkAppM ``impCongrCtx #[h₁, h₂]

def mkForallCongr (h : Expr) : MetaM Expr :=
  mkAppM ``forallCongr #[h]

/-- Return instance for `[Monad m]` if there is one -/
def isMonad? (m : Expr) : MetaM (Option Expr) :=
  try
    let monadType ← mkAppM `Monad #[m]
    let result    ← trySynthInstance monadType
    match result with
    | LOption.some inst => pure inst
    | _                 => pure none
  catch _ =>
    pure none

/-- Return `(n : type)`, a numeric literal of type `type`. The method fails if we don't have an instance `OfNat type n` -/
def mkNumeral (type : Expr) (n : Nat) : MetaM Expr := do
  let u ← getDecLevel type
  let inst ← synthInstance (mkApp2 (mkConst ``OfNat [u]) type (mkNatLit n))
  return mkApp3 (mkConst ``OfNat.ofNat [u]) type (mkNatLit n) inst

/--
  Return `a op b`, where `op` has name `opName` and is implemented using the typeclass `className`.
  This method assumes `a` and `b` have the same type, and typeclass `className` is heterogeneous.
  Examples of supported clases: `HAdd`, `HSub`, `HMul`.
```

```
  We use heterogeneous operators to ensure we have a uniform representation.
  -/
private def mkBinaryOp (className : Name) (opName : Name) (a b : Expr) : MetaM Expr := do
  let aType ← inferType a
  let u ← getDecLevel aType
  let inst ← synthInstance (mkApp3 (mkConst className [u, u, u]) aType aType aType)
  return mkApp6 (mkConst opName [u, u, u]) aType aType aType inst a b

/-- Return `a + b` using a heterogeneous `+`. This method assumes `a` and `b` have the same type. -/
def mkAdd (a b : Expr) : MetaM Expr := mkBinaryOp ``HAdd ``HAdd.hAdd a b

/-- Return `a - b` using a heterogeneous `-`. This method assumes `a` and `b` have the same type. -/
def mkSub (a b : Expr) : MetaM Expr := mkBinaryOp ``HSub ``HSub.hSub a b

/-- Return `a * b` using a heterogeneous `*`. This method assumes `a` and `b` have the same type. -/
def mkMul (a b : Expr) : MetaM Expr := mkBinaryOp ``HMul ``HMul.hMul a b

builtin_initialize registerTraceClass `Meta.appBuilder

end Lean.Meta
// ::::::::::::::
// Basic.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Data.LOption
import Lean.Environment
import Lean.Class
import Lean.ReducibilityAttrs
import Lean.Util.Trace
import Lean.Util.RecDepth
import Lean.Util.PPExt
import Lean.Util.OccursCheck
import Lean.Compiler.InlineAttrs
import Lean.Meta.TransparencyMode
import Lean.Meta.DiscrTreeTypes
import Lean.Eval
import Lean.CoreM

/-
This module provides four (mutually dependent) goodies that are needed for building the elaborator and tactic frameworks.
1- Weak head normal form computation with support for metavariables and transparency modes.
2- Definitionally equality checking with support for metavariables (aka unification modulo definitional equality).
3- Type inference.
```

```
4- Type class resolution.

They are packed into the MetaM monad.
-/

namespace Lean.Meta

builtin_initialize isDefEqStuckExceptionId : InternalExceptionId ← registerInternalExceptionId `isDefEqStuck

structure Config where
  foApprox           : Bool := false
  ctxApprox          : Bool := false
  quasiPatternApprox : Bool := false
  /- When `constApprox` is set to true,
     we solve `?m t =?= c` using
     `?m := fun _ => c`
     when `?m t` is not a higher-order pattern and `c` is not an application as -/
  constApprox        : Bool := false
  /-
    When the following flag is set,
    `isDefEq` throws the exeption `Exeption.isDefEqStuck`
    whenever it encounters a constraint `?m ... =?= t` where
    `?m` is read only.
    This feature is useful for type class resolution where
    we may want to notify the caller that the TC problem may be solveable
    later after it assigns `?m`. -/
  isDefEqStuckEx     : Bool := false
  transparency       : TransparencyMode := TransparencyMode.default
  /- If zetaNonDep == false, then non dependent let-decls are not zeta expanded. -/
  zetaNonDep         : Bool := true
  /- When `trackZeta == true`, we store zetaFVarIds all free variables that have been zeta-expanded. -/
  trackZeta          : Bool := false
  unificationHints   : Bool := true

structure ParamInfo where
  implicit     : Bool      := false
  instImplicit : Bool      := false
  hasFwdDeps   : Bool      := false
  backDeps     : Array Nat := #[]
  deriving Inhabited

def ParamInfo.isExplicit (p : ParamInfo) : Bool :=
  !p.implicit && !p.instImplicit

structure FunInfo where
  paramInfo  : Array ParamInfo := #[]
  resultDeps : Array Nat       := #[]
```

```
structure InfoCacheKey where
  transparency : TransparencyMode
  expr         : Expr
  nargs?       : Option Nat
  deriving Inhabited, BEq

namespace InfoCacheKey
instance : Hashable InfoCacheKey :=
  ⟨fun ⟨transparency, expr, nargs⟩ => mixHash (hash transparency) <| mixHash (hash expr) (hash nargs)⟩
end InfoCacheKey

open Std (PersistentArray PersistentHashMap)

abbrev SynthInstanceCache := PersistentHashMap Expr (Option Expr)

abbrev InferTypeCache := PersistentExprStructMap Expr
abbrev FunInfoCache   := PersistentHashMap InfoCacheKey FunInfo
abbrev WhnfCache      := PersistentExprStructMap Expr
structure Cache where
  inferType     : InferTypeCache := {}
  funInfo       : FunInfoCache   := {}
  synthInstance : SynthInstanceCache := {}
  whnfDefault   : WhnfCache := {} -- cache for closed terms and `TransparencyMode.default`
  whnfAll       : WhnfCache := {} -- cache for closed terms and `TransparencyMode.all`
  deriving Inhabited

structure PostponedEntry where
  lhs       : Level
  rhs       : Level

structure State where
  mctx        : MetavarContext := {}
  cache       : Cache := {}
  /- When `trackZeta == true`, then any let-decl free variable that is zeta expansion performed by `MetaM` is stored in `zetaFVarIds`.
-/
  zetaFVarIds : NameSet := {}
  postponed   : PersistentArray PostponedEntry := {}
  deriving Inhabited

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

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

```
instance : Inhabited (MetaM α) where
  default := fun _ _ => arbitrary

instance : MonadLCtx MetaM where
  getLCtx := return (← read).lctx

instance : MonadMCtx MetaM where
  getMCtx    := return (← get).mctx
  modifyMCtx f := modify fun s => { s with mctx := f s.mctx }

instance : AddMessageContext MetaM where
  addMessageContext := addMessageContextFull

@[inline] def MetaM.run (x : MetaM α) (ctx : Context := {}) (s : State := {}) : CoreM (α × State) :=
  x ctx |>.run s

@[inline] def MetaM.run' (x : MetaM α) (ctx : Context := {}) (s : State := {}) : CoreM α :=
  Prod.fst <$> x.run ctx s

@[inline] def MetaM.toIO (x : MetaM α) (ctxCore : Core.Context) (sCore : Core.State) (ctx : Context := {}) (s : State := {}) : IO (α × Core.State × State) := do
  let ((a, s), sCore) ← (x.run ctx s).toIO ctxCore sCore
  pure (a, sCore, s)

instance [MetaEval α] : MetaEval (MetaM α) :=
  ⟨fun env opts x _ => MetaEval.eval env opts x.run' true⟩

protected def throwIsDefEqStuck {α} : MetaM α :=
  throw <| Exception.internal isDefEqStuckExceptionId

builtin_initialize
  registerTraceClass `Meta
  registerTraceClass `Meta.debug

@[inline] def liftMetaM [MonadLiftT MetaM m] (x : MetaM α) : m α :=
  liftM x

@[inline] def mapMetaM [MonadControlT MetaM m] [Monad m] (f : forall {α}, MetaM α → MetaM α) {α} (x : m α) : m α :=
  controlAt MetaM fun runInBase => f <| runInBase x

@[inline] def map1MetaM [MonadControlT MetaM m] [Monad m] (f : forall {α}, (β → MetaM α) → MetaM α) {α} (k : β → m α) : m α :=
  controlAt MetaM fun runInBase => f fun b => runInBase <| k b

@[inline] def map2MetaM [MonadControlT MetaM m] [Monad m] (f : forall {α}, (β → γ → MetaM α) → MetaM α) {α} (k : β → γ → m α) : m α :=
  controlAt MetaM fun runInBase => f fun b c => runInBase <| k b c

section Methods
```

```
variable [MonadControlT MetaM n] [Monad n]

@[inline] def modifyCache (f : Cache → Cache) : MetaM Unit :=
  modify fun ⟨mctx, cache, zetaFVarIds, postponed⟩ => ⟨mctx, f cache, zetaFVarIds, postponed⟩

@[inline] def modifyInferTypeCache (f : InferTypeCache → InferTypeCache) : MetaM Unit :=
  modifyCache fun ⟨ic, c1, c2, c3, c4⟩ => ⟨f ic, c1, c2, c3, c4⟩

def getLocalInstances : MetaM LocalInstances :=
  return (← read).localInstances

def getConfig : MetaM Config :=
  return (← read).config

def setMCtx (mctx : MetavarContext) : MetaM Unit :=
  modify fun s => { s with mctx := mctx }

def resetZetaFVarIds : MetaM Unit :=
  modify fun s => { s with zetaFVarIds := {} }

def getZetaFVarIds : MetaM NameSet :=
  return (← get).zetaFVarIds

def getPostponed : MetaM (PersistentArray PostponedEntry) :=
  return (← get).postponed

def setPostponed (postponed : PersistentArray PostponedEntry) : MetaM Unit :=
  modify fun s => { s with postponed := postponed }

@[inline] def modifyPostponed (f : PersistentArray PostponedEntry → PersistentArray PostponedEntry) : MetaM Unit :=
  modify fun s => { s with postponed := f s.postponed }

builtin_initialize whnfRef : IO.Ref (Expr → MetaM Expr) ← IO.mkRef fun _ => throwError "whnf implementation was not set"
builtin_initialize inferTypeRef : IO.Ref (Expr → MetaM Expr) ← IO.mkRef fun _ => throwError "inferType implementation was not set"
builtin_initialize isExprDefEqAuxRef : IO.Ref (Expr → Expr → MetaM Bool) ← IO.mkRef fun _ _ => throwError "isDefEq implementation was not set"
builtin_initialize synthPendingRef : IO.Ref (MVarId → MetaM Bool) ← IO.mkRef fun _ => pure false

def whnf (e : Expr) : MetaM Expr :=
  withIncRecDepth do (← whnfRef.get) e

def whnfForall (e : Expr) : MetaM Expr := do
  let e' ← whnf e
  if e'.isForall then pure e' else pure e

def inferType (e : Expr) : MetaM Expr :=
  withIncRecDepth do (← inferTypeRef.get) e
```

```
protected def isExprDefEqAux (t s : Expr) : MetaM Bool :=
  withIncRecDepth do (← isExprDefEqAuxRef.get) t s

protected def synthPending (mvarId : MVarId) : MetaM Bool :=
  withIncRecDepth do (← synthPendingRef.get) mvarId

-- withIncRecDepth for a monad `n` such that `[MonadControlT MetaM n]`
protected def withIncRecDepth {α} (x : n α) : n α :=
  mapMetaM (withIncRecDepth (m := MetaM)) x

private def mkFreshExprMVarAtCore
    (mvarId : MVarId) (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr) (kind : MetavarKind) (userName : Name)
(numScopeArgs : Nat) : MetaM Expr := do
  modifyMCtx fun mctx => mctx.addExprMVarDecl mvarId userName lctx localInsts type kind numScopeArgs;
  return mkMVar mvarId

def mkFreshExprMVarAt
    (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr)
    (kind : MetavarKind := MetavarKind.natural) (userName : Name := Name.anonymous) (numScopeArgs : Nat := 0)
    : MetaM Expr := do
  let mvarId ← mkFreshId
  mkFreshExprMVarAtCore mvarId lctx localInsts type kind userName numScopeArgs

def mkFreshLevelMVar : MetaM Level := do
  let mvarId ← mkFreshId
  modifyMCtx fun mctx => mctx.addLevelMVarDecl mvarId;
  return mkLevelMVar mvarId

private def mkFreshExprMVarCore (type : Expr) (kind : MetavarKind) (userName : Name) : MetaM Expr := do
  let lctx ← getLCtx
  let localInsts ← getLocalInstances
  mkFreshExprMVarAt lctx localInsts type kind userName

private def mkFreshExprMVarImpl (type? : Option Expr) (kind : MetavarKind) (userName : Name) : MetaM Expr :=
  match type? with
  | some type => mkFreshExprMVarCore type kind userName
  | none      => do
    let u ← mkFreshLevelMVar
    let type ← mkFreshExprMVarCore (mkSort u) MetavarKind.natural Name.anonymous
    mkFreshExprMVarCore type kind userName

def mkFreshExprMVar (type? : Option Expr) (kind := MetavarKind.natural) (userName := Name.anonymous) : MetaM Expr :=
  mkFreshExprMVarImpl type? kind userName

def mkFreshTypeMVar (kind := MetavarKind.natural) (userName := Name.anonymous) : MetaM Expr := do
  let u ← mkFreshLevelMVar
```

```
  mkFreshExprMVar (mkSort u) kind userName

/- Low-level version of `MkFreshExprMVar` which allows users to create/reserve a `mvarId` using `mkFreshId`, and then later create
   the metavar using this method. -/
private def mkFreshExprMVarWithIdCore (mvarId : MVarId) (type : Expr)
    (kind : MetavarKind := MetavarKind.natural) (userName : Name := Name.anonymous) (numScopeArgs : Nat := 0)
    : MetaM Expr := do
  let lctx ← getLCtx
  let localInsts ← getLocalInstances
  mkFreshExprMVarAtCore mvarId lctx localInsts type kind userName numScopeArgs

def mkFreshExprMVarWithId (mvarId : MVarId) (type? : Option Expr := none) (kind : MetavarKind := MetavarKind.natural) (userName := Name.anonymous) : MetaM Expr :=
  match type? with
  | some type => mkFreshExprMVarWithIdCore mvarId type kind userName
  | none      => do
    let u ← mkFreshLevelMVar
    let type ← mkFreshExprMVar (mkSort u)
    mkFreshExprMVarWithIdCore mvarId type kind userName

def getTransparency : MetaM TransparencyMode :=
  return (← getConfig).transparency

def shouldReduceAll : MetaM Bool :=
  return (← getTransparency) == TransparencyMode.all

def shouldReduceReducibleOnly : MetaM Bool :=
  return (← getTransparency) == TransparencyMode.reducible

def getMVarDecl (mvarId : MVarId) : MetaM MetavarDecl := do
  let mctx ← getMCtx
  match mctx.findDecl? mvarId with
  | some d => pure d
  | none   => throwError! "unknown metavariable '{mkMVar mvarId}'"

def setMVarKind (mvarId : MVarId) (kind : MetavarKind) : MetaM Unit :=
  modifyMCtx fun mctx => mctx.setMVarKind mvarId kind

/- Update the type of the given metavariable. This function assumes the new type is
   definitionally equal to the current one -/
def setMVarType (mvarId : MVarId) (type : Expr) : MetaM Unit := do
  modifyMCtx fun mctx => mctx.setMVarType mvarId type

def isReadOnlyExprMVar (mvarId : MVarId) : MetaM Bool := do
  let mvarDecl ← getMVarDecl mvarId
  let mctx     ← getMCtx
  return mvarDecl.depth != mctx.depth
```

```
def isReadOnlyOrSyntheticOpaqueExprMVar (mvarId : MVarId) : MetaM Bool := do
  let mvarDecl ← getMVarDecl mvarId
  match mvarDecl.kind with
  | MetavarKind.syntheticOpaque => pure true
  | _ =>
    let mctx ← getMCtx
    return mvarDecl.depth != mctx.depth

def isReadOnlyLevelMVar (mvarId : MVarId) : MetaM Bool := do
  let mctx ← getMCtx
  match mctx.findLevelDepth? mvarId with
  | some depth => return depth != mctx.depth
  | _          => throwError! "unknown universe metavariable '{mkLevelMVar mvarId}'"

def renameMVar (mvarId : MVarId) (newUserName : Name) : MetaM Unit :=
  modifyMCtx fun mctx => mctx.renameMVar mvarId newUserName

def isExprMVarAssigned (mvarId : MVarId) : MetaM Bool :=
  return (← getMCtx).isExprAssigned mvarId

def getExprMVarAssignment? (mvarId : MVarId) : MetaM (Option Expr) :=
  return (← getMCtx).getExprAssignment? mvarId

/-- Return true if `e` contains `mvarId` directly or indirectly -/
def occursCheck (mvarId : MVarId) (e : Expr) : MetaM Bool :=
  return (← getMCtx).occursCheck mvarId e

def assignExprMVar (mvarId : MVarId) (val : Expr) : MetaM Unit :=
  modifyMCtx fun mctx => mctx.assignExpr mvarId val

def isDelayedAssigned (mvarId : MVarId) : MetaM Bool :=
  return (← getMCtx).isDelayedAssigned mvarId

def getDelayedAssignment? (mvarId : MVarId) : MetaM (Option DelayedMetavarAssignment) :=
  return (← getMCtx).getDelayedAssignment? mvarId

def hasAssignableMVar (e : Expr) : MetaM Bool :=
  return (← getMCtx).hasAssignableMVar e

def throwUnknownFVar {α} (fvarId : FVarId) : MetaM α :=
  throwError! "unknown free variable '{mkFVar fvarId}'"

def findLocalDecl? (fvarId : FVarId) : MetaM (Option LocalDecl) :=
  return (← getLCtx).find? fvarId

def getLocalDecl (fvarId : FVarId) : MetaM LocalDecl := do
```

```
  match (← getLCtx).find? fvarId with
  | some d => pure d
  | none   => throwUnknownFVar fvarId

def getFVarLocalDecl (fvar : Expr) : MetaM LocalDecl :=
  getLocalDecl fvar.fvarId!

def getLocalDeclFromUserName (userName : Name) : MetaM LocalDecl := do
  match (← getLCtx).findFromUserName? userName with
  | some d => pure d
  | none   => throwError! "unknown local declaration '{userName}'"

def instantiateLevelMVars (u : Level) : MetaM Level :=
  MetavarContext.instantiateLevelMVars u

def instantiateMVars (e : Expr) : MetaM Expr :=
  (MetavarContext.instantiateExprMVars e).run

def instantiateLocalDeclMVars (localDecl : LocalDecl) : MetaM LocalDecl := do
  match localDecl with
  | LocalDecl.cdecl idx id n type bi  =>
    let type ← instantiateMVars type
    return LocalDecl.cdecl idx id n type bi
  | LocalDecl.ldecl idx id n type val nonDep =>
    let type ← instantiateMVars type
    let val ← instantiateMVars val
    return LocalDecl.ldecl idx id n type val nonDep

@[inline] def liftMkBindingM {α} (x : MetavarContext.MkBindingM α) : MetaM α := do
  match x (← getLCtx) { mctx := (← getMCtx), ngen := (← getNGen) } with
  | EStateM.Result.ok e newS => do
    setNGen newS.ngen;
    setMCtx newS.mctx;
    pure e
  | EStateM.Result.error (MetavarContext.MkBinding.Exception.revertFailure mctx lctx toRevert decl) newS => do
    setMCtx newS.mctx;
    setNGen newS.ngen;
    throwError "failed to create binder due to failure when reverting variable dependencies"

def mkForallFVars (xs : Array Expr) (e : Expr) (usedOnly : Bool := false) (usedLetOnly : Bool := true) : MetaM Expr :=
  if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.mkForall xs e usedOnly usedLetOnly

def mkLambdaFVars (xs : Array Expr) (e : Expr) (usedOnly : Bool := false) (usedLetOnly : Bool := true) : MetaM Expr :=
  if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.mkLambda xs e usedOnly usedLetOnly

def mkLetFVars (xs : Array Expr) (e : Expr) : MetaM Expr :=
  mkLambdaFVars xs e
```

```
def mkArrow (d b : Expr) : MetaM Expr := do
  let n ← mkFreshUserName `x
  return Lean.mkForall n BinderInfo.default d b

def elimMVarDeps (xs : Array Expr) (e : Expr) (preserveOrder : Bool := false) : MetaM Expr :=
  if xs.isEmpty then pure e else liftMkBindingM <| MetavarContext.elimMVarDeps xs e preserveOrder

@[inline] def withConfig {α} (f : Config → Config) : n α → n α :=
  mapMetaM <| withReader (fun ctx => { ctx with config := f ctx.config })

@[inline] def withTrackingZeta {α} (x : n α) : n α :=
  withConfig (fun cfg => { cfg with trackZeta := true }) x

@[inline] def withTransparency {α} (mode : TransparencyMode) : n α → n α :=
  mapMetaM <| withConfig (fun config => { config with transparency := mode })

@[inline] def withDefault {α} (x : n α) : n α :=
  withTransparency TransparencyMode.default x

@[inline] def withReducible {α} (x : n α) : n α :=
  withTransparency TransparencyMode.reducible x

@[inline] def withReducibleAndInstances {α} (x : n α) : n α :=
  withTransparency TransparencyMode.instances x

@[inline] def withAtLeastTransparency {α} (mode : TransparencyMode) (x : n α) : n α :=
  withConfig
    (fun config =>
      let oldMode := config.transparency
      let mode    := if oldMode.lt mode then mode else oldMode
      { config with transparency := mode })
    x

/-- Save cache, execute `x`, restore cache -/
@[inline] private def savingCacheImpl {α} (x : MetaM α) : MetaM α := do
  let s ← get
  let savedCache := s.cache
  try x finally modify fun s => { s with cache := savedCache }

@[inline] def savingCache {α} : n α → n α :=
  mapMetaM savingCacheImpl

def getTheoremInfo (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
  if (← shouldReduceAll) then
    return some info
  else
```

```
    return none

private def getDefInfoTemp (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
  match (← getTransparency) with
  | TransparencyMode.all => return some info
  | TransparencyMode.default => return some info
  | _ =>
    if (← isReducible info.name) then
      return some info
    else
      return none

/- Remark: we later define `getConst?` at `GetConst.lean` after we define `Instances.lean`.
   This method is only used to implement `isClassQuickConst?`.
   It is very similar to `getConst?`, but it returns none when `TransparencyMode.instances` and
   `constName` is an instance. This difference should be irrelevant for `isClassQuickConst?`. -/
private def getConstTemp? (constName : Name) : MetaM (Option ConstantInfo) := do
  let env ← getEnv
  match env.find? constName with
  | some (info@(ConstantInfo.thmInfo _))  => getTheoremInfo info
  | some (info@(ConstantInfo.defnInfo _)) => getDefInfoTemp info
  | some info                             => pure (some info)
  | none                                  => throwUnknownConstant constName

private def isClassQuickConst? (constName : Name) : MetaM (LOption Name) := do
  let env ← getEnv
  if isClass env constName then
    pure (LOption.some constName)
  else
    match (← getConstTemp? constName) with
    | some _ => pure LOption.undef
    | none   => pure LOption.none

private partial def isClassQuick? : Expr → MetaM (LOption Name)
  | Expr.bvar ..         => pure LOption.none
  | Expr.lit ..          => pure LOption.none
  | Expr.fvar ..         => pure LOption.none
  | Expr.sort ..         => pure LOption.none
  | Expr.lam ..          => pure LOption.none
  | Expr.letE ..         => pure LOption.undef
  | Expr.proj ..         => pure LOption.undef
  | Expr.forallE _ _ b _ => isClassQuick? b
  | Expr.mdata _ e _     => isClassQuick? e
  | Expr.const n _ _     => isClassQuickConst? n
  | Expr.mvar mvarId _   => do
    match (← getExprMVarAssignment? mvarId) with
    | some val => isClassQuick? val
```

```
    | none     => pure LOption.none
  | Expr.app f _ _       =>
    match f.getAppFn with
    | Expr.const n .. => isClassQuickConst? n
    | Expr.lam ..     => pure LOption.undef
    | _              => pure LOption.none

def saveAndResetSynthInstanceCache : MetaM SynthInstanceCache := do
  let s ← get
  let savedSythInstance := s.cache.synthInstance
  modifyCache fun c => { c with synthInstance := {} }
  pure savedSythInstance

def restoreSynthInstanceCache (cache : SynthInstanceCache) : MetaM Unit :=
  modifyCache fun c => { c with synthInstance := cache }

@[inline] private def resettingSynthInstanceCacheImpl {α} (x : MetaM α) : MetaM α := do
  let savedSythInstance ← saveAndResetSynthInstanceCache
  try x finally restoreSynthInstanceCache savedSythInstance

/-- Reset `synthInstance` cache, execute `x`, and restore cache -/
@[inline] def resettingSynthInstanceCache {α} : n α → n α :=
  mapMetaM resettingSynthInstanceCacheImpl

@[inline] def resettingSynthInstanceCacheWhen {α} (b : Bool) (x : n α) : n α :=
  if b then resettingSynthInstanceCache x else x

private def withNewLocalInstanceImp {α} (className : Name) (fvar : Expr) (k : MetaM α) : MetaM α := do
  let localDecl ← getFVarLocalDecl fvar
  /- Recall that we use `auxDecl` binderInfo when compiling recursive declarations. -/
  match localDecl.binderInfo with
  | BinderInfo.auxDecl => k
  | _ =>
    resettingSynthInstanceCache <|
      withReader
        (fun ctx => { ctx with localInstances := ctx.localInstances.push { className := className, fvar := fvar } })
        k

/-- Add entry `{ className := className, fvar := fvar }` to localInstances,
    and then execute continuation `k`.
    It resets the type class cache using `resettingSynthInstanceCache`. -/
def withNewLocalInstance {α} (className : Name) (fvar : Expr) : n α → n α :=
  mapMetaM <| withNewLocalInstanceImp className fvar

private def fvarsSizeLtMaxFVars (fvars : Array Expr) (maxFVars? : Option Nat) : Bool :=
  match maxFVars? with
  | some maxFVars => fvars.size < maxFVars
```

```
  | none          => true

mutual
  /--
    `withNewLocalInstances isClassExpensive fvars j k` updates the vector or local instances
    using free variables `fvars[j] ... fvars.back`, and execute `k`.

    - `isClassExpensive` is defined later.
    - The type class chache is reset whenever a new local instance is found.
    - `isClassExpensive` uses `whnf` which depends (indirectly) on the set of local instances.
      Thus, each new local instance requires a new `resettingSynthInstanceCache`. -/
  private partial def withNewLocalInstancesImp {α}
      (fvars : Array Expr) (i : Nat) (k : MetaM α) : MetaM α := do
    if h : i < fvars.size then
      let fvar := fvars.get ⟨i, h⟩
      let decl ← getFVarLocalDecl fvar
      match (← isClassQuick? decl.type) with
      | LOption.none   => withNewLocalInstancesImp fvars (i+1) k
      | LOption.undef  =>
        match (← isClassExpensive? decl.type) with
        | none   => withNewLocalInstancesImp fvars (i+1) k
        | some c => withNewLocalInstance c fvar <| withNewLocalInstancesImp fvars (i+1) k
      | LOption.some c => withNewLocalInstance c fvar <| withNewLocalInstancesImp fvars (i+1) k
    else
      k

  /--
    `forallTelescopeAuxAux lctx fvars j type`
    Remarks:
    - `lctx` is the `MetaM` local context extended with declarations for `fvars`.
    - `type` is the type we are computing the telescope for. It contains only
      dangling bound variables in the range `[j, fvars.size)`
    - if `reducing? == true` and `type` is not `forallE`, we use `whnf`.
    - when `type` is not a `forallE` nor it can't be reduced to one, we
      excute the continuation `k`.

    Here is an example that demonstrates the `reducing?`.
    Suppose we have
    ```
    abbrev StateM s a := s -> Prod a s
    ```
    Now, assume we are trying to build the telescope for
    ```
    forall (x : Nat), StateM Int Bool
    ```
    if `reducing == true`, the function executes `k #[(x : Nat) (s : Int)] Bool`.
    if `reducing == false`, the function executes `k #[(x : Nat)] (StateM Int Bool)`
```

```

  if `maxFVars?` is `some max`, then we interrupt the telescope construction
  when `fvars.size == max`
-/
private partial def forallTelescopeReducingAuxAux {α}
    (reducing          : Bool) (maxFVars? : Option Nat)
    (type              : Expr)
    (k                 : Array Expr → Expr → MetaM α) : MetaM α := do
  let rec process (lctx : LocalContext) (fvars : Array Expr) (j : Nat) (type : Expr) : MetaM α := do
    match type with
    | Expr.forallE n d b c =>
      if fvarsSizeLtMaxFVars fvars maxFVars? then
        let d     := d.instantiateRevRange j fvars.size fvars
        let fvarId ← mkFreshId
        let lctx  := lctx.mkLocalDecl fvarId n d c.binderInfo
        let fvar  := mkFVar fvarId
        let fvars := fvars.push fvar
        process lctx fvars j b
      else
        let type := type.instantiateRevRange j fvars.size fvars;
        withReader (fun ctx => { ctx with lctx := lctx }) do
          withNewLocalInstancesImp fvars j do
            k fvars type
    | _ =>
      let type := type.instantiateRevRange j fvars.size fvars;
      withReader (fun ctx => { ctx with lctx := lctx }) do
        withNewLocalInstancesImp fvars j do
          if reducing && fvarsSizeLtMaxFVars fvars maxFVars? then
            let newType ← whnf type
            if newType.isForall then
              process lctx fvars fvars.size newType
            else
              k fvars type
          else
            k fvars type
  process (← getLCtx) #[] 0 type

private partial def forallTelescopeReducingAux {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → MetaM α) : MetaM α := do
  match maxFVars? with
  | some 0 => k #[] type
  | _ => do
    let newType ← whnf type
    if newType.isForall then
      forallTelescopeReducingAuxAux true maxFVars? newType k
    else
      k #[] type
```

```
  private partial def isClassExpensive? : Expr → MetaM (Option Name)
    | type => withReducible <| -- when testing whether a type is a type class, we only unfold reducible constants.
      forallTelescopeReducingAux type none fun xs type => do
        let env ← getEnv
        match type.getAppFn with
        | Expr.const c _ _ => do
          if isClass env c then
            return some c
          else
            -- make sure abbreviations are unfolded
            match (← whnf type).getAppFn with
            | Expr.const c _ _ => return if isClass env c then some c else none
            | _ => return none
        | _ => return none

  private partial def isClassImp? (type : Expr) : MetaM (Option Name) := do
    match (← isClassQuick? type) with
    | LOption.none   => pure none
    | LOption.some c => pure (some c)
    | LOption.undef  => isClassExpensive? type

end

def isClass? (type : Expr) : MetaM (Option Name) :=
  try isClassImp? type catch _ => pure none

private def withNewLocalInstancesImpAux {α} (fvars : Array Expr) (j : Nat) : n α → n α :=
  mapMetaM <| withNewLocalInstancesImp fvars j

partial def withNewLocalInstances {α} (fvars : Array Expr) (j : Nat) : n α → n α :=
  mapMetaM <| withNewLocalInstancesImpAux fvars j

@[inline] private def forallTelescopeImp {α} (type : Expr) (k : Array Expr → Expr → MetaM α) : MetaM α := do
  forallTelescopeReducingAuxAux (reducing := false) (maxFVars? := none) type k

/--
  Given `type` of the form `forall xs, A`, execute `k xs A`.
  This combinator will declare local declarations, create free variables for them,
  execute `k` with updated local context, and make sure the cache is restored after executing `k`. -/
def forallTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
  map2MetaM (fun k => forallTelescopeImp type k) k

private def forallTelescopeReducingImp {α} (type : Expr) (k : Array Expr → Expr → MetaM α) : MetaM α :=
  forallTelescopeReducingAux type (maxFVars? := none) k

/--
```

```
  Similar to `forallTelescope`, but given `type` of the form `forall xs, A`,
  it reduces `A` and continues bulding the telescope if it is a `forall`. -/
def forallTelescopeReducing {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
  map2MetaM (fun k => forallTelescopeReducingImp type k) k

private def forallBoundedTelescopeImp {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → MetaM α) : MetaM α :=
  forallTelescopeReducingAux type maxFVars? k

/--
  Similar to `forallTelescopeReducing`, stops constructing the telescope when
  it reaches size `maxFVars`. -/
def forallBoundedTelescope {α} (type : Expr) (maxFVars? : Option Nat) (k : Array Expr → Expr → n α) : n α :=
  map2MetaM (fun k => forallBoundedTelescopeImp type maxFVars? k) k

/-- Similar to `forallTelescopeAuxAux` but for lambda and let expressions. -/
private partial def lambdaTelescopeAux {α}
    (k : Array Expr → Expr → MetaM α)
    : Bool → LocalContext → Array Expr → Nat → Expr → MetaM α
  | consumeLet, lctx, fvars, j, Expr.lam n d b c => do
    let d := d.instantiateRevRange j fvars.size fvars
    let fvarId ← mkFreshId
    let lctx := lctx.mkLocalDecl fvarId n d c.binderInfo
    let fvar := mkFVar fvarId
    lambdaTelescopeAux k consumeLet lctx (fvars.push fvar) j b
  | true, lctx, fvars, j, Expr.letE n t v b _ => do
    let t := t.instantiateRevRange j fvars.size fvars
    let v := v.instantiateRevRange j fvars.size fvars
    let fvarId ← mkFreshId
    let lctx := lctx.mkLetDecl fvarId n t v
    let fvar := mkFVar fvarId
    lambdaTelescopeAux k true lctx (fvars.push fvar) j b
  | _, lctx, fvars, j, e =>
    let e := e.instantiateRevRange j fvars.size fvars;
    withReader (fun ctx => { ctx with lctx := lctx }) do
      withNewLocalInstancesImp fvars j do
        k fvars e

private partial def lambdaTelescopeImp {α} (e : Expr) (consumeLet : Bool) (k : Array Expr → Expr → MetaM α) : MetaM α := do
  let rec process (consumeLet : Bool) (lctx : LocalContext) (fvars : Array Expr) (j : Nat) (e : Expr) : MetaM α := do
    match consumeLet, e with
    | _, Expr.lam n d b c =>
      let d := d.instantiateRevRange j fvars.size fvars
      let fvarId ← mkFreshId
      let lctx := lctx.mkLocalDecl fvarId n d c.binderInfo
      let fvar := mkFVar fvarId
      process consumeLet lctx (fvars.push fvar) j b
    | true, Expr.letE n t v b _ => do
```

```
      let t := t.instantiateRevRange j fvars.size fvars
      let v := v.instantiateRevRange j fvars.size fvars
      let fvarId ← mkFreshId
      let lctx := lctx.mkLetDecl fvarId n t v
      let fvar := mkFVar fvarId
      process true lctx (fvars.push fvar) j b
    | _, e =>
      let e := e.instantiateRevRange j fvars.size fvars
      withReader (fun ctx => { ctx with lctx := lctx }) do
        withNewLocalInstancesImp fvars j do
          k fvars e
  process consumeLet (← getLCtx) #[] 0 e

/-- Similar to `forallTelescope` but for lambda and let expressions. -/
def lambdaLetTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
  map2MetaM (fun k => lambdaTelescopeImp type true k) k

/-- Similar to `forallTelescope` but for lambda expressions. -/
def lambdaTelescope {α} (type : Expr) (k : Array Expr → Expr → n α) : n α :=
  map2MetaM (fun k => lambdaTelescopeImp type false k) k

/-- Return the parameter names for the givel global declaration. -/
def getParamNames (declName : Name) : MetaM (Array Name) := do
  let cinfo ← getConstInfo declName
  forallTelescopeReducing cinfo.type fun xs _ => do
    xs.mapM fun x => do
      let localDecl ← getLocalDecl x.fvarId!
      pure localDecl.userName

-- `kind` specifies the metavariable kind for metavariables not corresponding to instance implicit `[ ... ]` arguments.
private partial def forallMetaTelescopeReducingAux
    (e : Expr) (reducing : Bool) (maxMVars? : Option Nat) (kind : MetavarKind) : MetaM (Array Expr × Array BinderInfo × Expr) :=
  let rec process (mvars : Array Expr) (bis : Array BinderInfo) (j : Nat) (type : Expr) : MetaM (Array Expr × Array BinderInfo × Expr) := do
    match type with
    | Expr.forallE n d b c =>
      let cont : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
        let d  := d.instantiateRevRange j mvars.size mvars
        let k  := if c.binderInfo.isInstImplicit then  MetavarKind.synthetic else kind
        let mvar ← mkFreshExprMVar d k n
        let mvars := mvars.push mvar
        let bis   := bis.push c.binderInfo
        process mvars bis j b
      match maxMVars? with
      | none          => cont ()
      | some maxMVars =>
        if mvars.size < maxMVars then
```

```
          cont ()
        else
          let type := type.instantiateRevRange j mvars.size mvars;
          pure (mvars, bis, type)
    | _ =>
      let type := type.instantiateRevRange j mvars.size mvars;
      if reducing then do
        let newType ← whnf type;
        if newType.isForall then
          process mvars bis mvars.size newType
        else
          pure (mvars, bis, type)
      else
        pure (mvars, bis, type)
  process #[] #[] 0 e

/-- Similar to `forallTelescope`, but creates metavariables instead of free variables. -/
def forallMetaTelescope (e : Expr) (kind := MetavarKind.natural) : MetaM (Array Expr × Array BinderInfo × Expr) :=
  forallMetaTelescopeReducingAux e (reducing := false) (maxMVars? := none) kind

/-- Similar to `forallTelescopeReducing`, but creates metavariables instead of free variables. -/
def forallMetaTelescopeReducing (e : Expr) (maxMVars? : Option Nat := none) (kind := MetavarKind.natural) : MetaM (Array Expr × Array BinderInfo × Expr) :=
  forallMetaTelescopeReducingAux e (reducing := true) maxMVars? kind

/-- Similar to `forallMetaTelescopeReducingAux` but for lambda expressions. -/
partial def lambdaMetaTelescope (e : Expr) (maxMVars? : Option Nat := none) : MetaM (Array Expr × Array BinderInfo × Expr) :=
  let rec process (mvars : Array Expr) (bis : Array BinderInfo) (j : Nat) (type : Expr) : MetaM (Array Expr × Array BinderInfo × Expr) := do
    let finalize : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
      let type := type.instantiateRevRange j mvars.size mvars
      pure (mvars, bis, type)
    let cont : Unit → MetaM (Array Expr × Array BinderInfo × Expr) := fun _ => do
      match type with
      | Expr.lam n d b c =>
        let d     := d.instantiateRevRange j mvars.size mvars
        let mvar ← mkFreshExprMVar d
        let mvars := mvars.push mvar
        let bis   := bis.push c.binderInfo
        process mvars bis j b
      | _ => finalize ()
    match maxMVars? with
    | none          => cont ()
    | some maxMVars =>
      if mvars.size < maxMVars then
        cont ()
      else
```

```
        finalize ()
  process #[] #[] 0 e

private def withNewFVar {α} (fvar fvarType : Expr) (k : Expr → MetaM α) : MetaM α := do
  match (← isClass? fvarType) with
  | none   => k fvar
  | some c => withNewLocalInstance c fvar <| k fvar

private def withLocalDeclImp {α} (n : Name) (bi : BinderInfo) (type : Expr) (k : Expr → MetaM α) : MetaM α := do
  let fvarId ← mkFreshId
  let ctx ← read
  let lctx := ctx.lctx.mkLocalDecl fvarId n type bi
  let fvar := mkFVar fvarId
  withReader (fun ctx => { ctx with lctx := lctx }) do
    withNewFVar fvar type k

def withLocalDecl {α} (name : Name) (bi : BinderInfo) (type : Expr) (k : Expr → n α) : n α :=
  map1MetaM (fun k => withLocalDeclImp name bi type k) k

def withLocalDeclD {α} (name : Name) (type : Expr) (k : Expr → n α) : n α :=
  withLocalDecl name BinderInfo.default type k

private def withLetDeclImp {α} (n : Name) (type : Expr) (val : Expr) (k : Expr → MetaM α) : MetaM α := do
  let fvarId ← mkFreshId
  let ctx ← read
  let lctx := ctx.lctx.mkLetDecl fvarId n type val
  let fvar := mkFVar fvarId
  withReader (fun ctx => { ctx with lctx := lctx }) do
    withNewFVar fvar type k

def withLetDecl {α} (name : Name) (type : Expr) (val : Expr) (k : Expr → n α) : n α :=
  map1MetaM (fun k => withLetDeclImp name type val k) k

private def withExistingLocalDeclsImp {α} (decls : List LocalDecl) (k : MetaM α) : MetaM α := do
  let ctx ← read
  let numLocalInstances := ctx.localInstances.size
  let lctx := decls.foldl (fun (lctx : LocalContext) decl => lctx.addDecl decl) ctx.lctx
  withReader (fun ctx => { ctx with lctx := lctx }) do
    let newLocalInsts ← decls.foldlM
      (fun (newlocalInsts : Array LocalInstance) (decl : LocalDecl) => (do {
        match (← isClass? decl.type) with
        | none   => pure newlocalInsts
        | some c => pure <| newlocalInsts.push { className := c, fvar := decl.toExpr } } : MetaM _))
      ctx.localInstances;
    if newLocalInsts.size == numLocalInstances then
      k
    else
```

```
      resettingSynthInstanceCache <| withReader (fun ctx => { ctx with localInstances := newLocalInsts }) k

def withExistingLocalDecls {α} (decls : List LocalDecl) : n α → n α :=
  mapMetaM <| withExistingLocalDeclsImp decls

private def withNewMCtxDepthImp {α} (x : MetaM α) : MetaM α := do
  let s ← get
  let savedMCtx  := s.mctx
  modifyMCtx fun mctx => mctx.incDepth
  try x finally setMCtx savedMCtx

/--
  Save cache and `MetavarContext`, bump the `MetavarContext` depth, execute `x`,
  and restore saved data. -/
def withNewMCtxDepth {α} : n α → n α :=
  mapMetaM withNewMCtxDepthImp

private def withLocalContextImp {α} (lctx : LocalContext) (localInsts : LocalInstances) (x : MetaM α) : MetaM α := do
  let localInstsCurr ← getLocalInstances
  withReader (fun ctx => { ctx with lctx := lctx, localInstances := localInsts }) do
    if localInsts == localInstsCurr then
      x
    else
      resettingSynthInstanceCache x

def withLCtx {α} (lctx : LocalContext) (localInsts : LocalInstances) : n α → n α :=
  mapMetaM <| withLocalContextImp lctx localInsts

private def withMVarContextImp {α} (mvarId : MVarId) (x : MetaM α) : MetaM α := do
  let mvarDecl ← getMVarDecl mvarId
  withLocalContextImp mvarDecl.lctx mvarDecl.localInstances x

/--
  Execute `x` using the given metavariable `LocalContext` and `LocalInstances`.
  The type class resolution cache is flushed when executing `x` if its `LocalInstances` are
  different from the current ones. -/
def withMVarContext {α} (mvarId : MVarId) : n α → n α :=
  mapMetaM <| withMVarContextImp mvarId

private def withMCtxImp {α} (mctx : MetavarContext) (x : MetaM α) : MetaM α := do
  let mctx' ← getMCtx
  setMCtx mctx
  try x finally setMCtx mctx'

def withMCtx {α} (mctx : MetavarContext) : n α → n α :=
  mapMetaM <| withMCtxImp mctx
```

```
@[inline] private def approxDefEqImp {α} (x : MetaM α) : MetaM α :=
  withConfig (fun config => { config with foApprox := true, ctxApprox := true, quasiPatternApprox := true}) x

/-- Execute `x` using approximate unification: `foApprox`, `ctxApprox` and `quasiPatternApprox`.  -/
@[inline] def approxDefEq {α} : n α → n α :=
  mapMetaM approxDefEqImp

@[inline] private def fullApproxDefEqImp {α} (x : MetaM α) : MetaM α :=
  withConfig (fun config => { config with foApprox := true, ctxApprox := true, quasiPatternApprox := true, constApprox := true }) x

/--
  Similar to `approxDefEq`, but uses all available approximations.
  We don't use `constApprox` by default at `approxDefEq` because it often produces undesirable solution for monadic code.
  For example, suppose we have `pure (x > 0)` which has type `?m Prop`. We also have the goal `[Pure ?m]`.
  Now, assume the expected type is `IO Bool`. Then, the unification constraint `?m Prop =?= IO Bool` could be solved
  as `?m := fun _ => IO Bool` using `constApprox`, but this spurious solution would generate a failure when we try to
  solve `[Pure (fun _ => IO Bool)]` -/
@[inline] def fullApproxDefEq {α} : n α → n α :=
  mapMetaM fullApproxDefEqImp

def normalizeLevel (u : Level) : MetaM Level := do
  let u ← instantiateLevelMVars u
  pure u.normalize

def assignLevelMVar (mvarId : MVarId) (u : Level) : MetaM Unit := do
  modifyMCtx fun mctx => mctx.assignLevel mvarId u

def whnfR (e : Expr) : MetaM Expr :=
  withTransparency TransparencyMode.reducible <| whnf e

def whnfD (e : Expr) : MetaM Expr :=
  withTransparency TransparencyMode.default <| whnf e

def whnfI (e : Expr) : MetaM Expr :=
  withTransparency TransparencyMode.instances <| whnf e

def setInlineAttribute (declName : Name) (kind := Compiler.InlineAttributeKind.inline): MetaM Unit := do
  let env ← getEnv
  match Compiler.setInlineAttribute env declName kind with
  | Except.ok env    => setEnv env
  | Except.error msg => throwError msg

private partial def instantiateForallAux (ps : Array Expr) (i : Nat) (e : Expr) : MetaM Expr := do
  if h : i < ps.size then
    let p := ps.get ⟨i, h⟩
    let e ← whnf e
    match e with
```

```
    | Expr.forallE _ _ b _ => instantiateForallAux ps (i+1) (b.instantiate1 p)
    | _                    => throwError "invalid instantiateForall, too many parameters"
  else
    pure e

/- 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]`. -/
def instantiateForall (e : Expr) (ps : Array Expr) : MetaM Expr :=
  instantiateForallAux ps 0 e

private partial def instantiateLambdaAux (ps : Array Expr) (i : Nat) (e : Expr) : MetaM Expr := do
  if h : i < ps.size then
    let p := ps.get ⟨i, h⟩
    let e ← whnf e
    match e with
    | Expr.lam _ _ b _ => instantiateLambdaAux ps (i+1) (b.instantiate1 p)
    | _                => throwError "invalid instantiateLambda, too many parameters"
  else
    pure e

/- 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 -/
def instantiateLambda (e : Expr) (ps : Array Expr) : MetaM Expr :=
  instantiateLambdaAux ps 0 e

/-- Return true iff `e` depends on the free variable `fvarId` -/
def dependsOn (e : Expr) (fvarId : FVarId) : MetaM Bool :=
  return (← getMCtx).exprDependsOn e fvarId

def ppExpr (e : Expr) : MetaM Format := do
  let env  ← getEnv
  let mctx ← getMCtx
  let lctx ← getLCtx
  let opts ← getOptions
  let ctxCore  ← readThe Core.Context
  Lean.ppExpr { env := env, mctx := mctx, lctx := lctx, opts := opts, currNamespace := ctxCore.currNamespace, openDecls := ctxCore.openDecls  } e

@[inline] protected def orelse {α} (x y : MetaM α) : MetaM α := do
  let env  ← getEnv
  let mctx ← getMCtx
  try x catch _ => setEnv env; setMCtx mctx; y

instance {α} : OrElse (MetaM α) := ⟨Meta.orelse⟩

@[inline] private def orelseMergeErrorsImp {α} (x y : MetaM α)
```

```
    (mergeRef : Syntax → Syntax → Syntax := fun r₁ r₂ => r₁)
    (mergeMsg : MessageData → MessageData → MessageData := fun m₁ m₂ => m₁ ++ Format.line ++ m₂) : MetaM α := do
  let env  ← getEnv
  let mctx ← getMCtx
  try
    x
  catch ex =>
    setEnv env
    setMCtx mctx
    match ex with
    | Exception.error ref₁ m₁ =>
      try
        y
      catch
        | Exception.error ref₂ m₂ => throw <| Exception.error (mergeRef ref₁ ref₂) (mergeMsg m₁ m₂)
        | ex => throw ex
    | ex => throw ex

/--
  Similar to `orelse`, but merge errors. Note that internal errors are not caught.
  The default `mergeRef` uses the `ref` (position information) for the first message.
  The default `mergeMsg` combines error messages using `Format.line ++ Format.line` as a separator. -/
@[inline] def orelseMergeErrors {α m} [MonadControlT MetaM m] [Monad m] (x y : m α)
    (mergeRef : Syntax → Syntax → Syntax := fun r₁ r₂ => r₁)
    (mergeMsg : MessageData → MessageData → MessageData := fun m₁ m₂ => m₁ ++ Format.line ++ Format.line ++ m₂) : m α := do
  controlAt MetaM fun runInBase => orelseMergeErrorsImp (runInBase x) (runInBase y) mergeRef mergeMsg

/-- Execute `x`, and apply `f` to the produced error message -/
def mapErrorImp {α} (x : MetaM α) (f : MessageData → MessageData) : MetaM α := do
  try
    x
  catch
    | Exception.error ref msg => throw <| Exception.error ref <| f msg
    | ex => throw ex

@[inline] def mapError {α m} [MonadControlT MetaM m] [Monad m] (x : m α) (f : MessageData → MessageData) : m α :=
  controlAt MetaM fun runInBase => mapErrorImp (runInBase x) f

/-- `commitWhenSome? x` executes `x` and keep modifications when it returns `some a`. -/
@[specialize] def commitWhenSome? {α} (x? : MetaM (Option α)) : MetaM (Option α) := do
  let env  ← getEnv
  let mctx ← getMCtx
  try
    match (← x?) with
    | some a => pure (some a)
    | none   =>
      setEnv env
```

```
      setMCtx mctx
      pure none
  catch ex =>
    setEnv env
    setMCtx mctx
    throw ex

end Methods
end Meta

export Meta (MetaM)

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

/-
This is not the Kernel type checker, but an auxiliary method for checking
whether terms produced by tactics and `isDefEq` are type correct.
-/

namespace Lean.Meta

private def ensureType (e : Expr) : MetaM Unit := do
  discard <| getLevel e

def throwLetTypeMismatchMessage {α} (fvarId : FVarId) : MetaM α := do
  let lctx ← getLCtx
  match lctx.find? fvarId with
  | some (LocalDecl.ldecl _ _ n t v _) => do
    let vType ← inferType v
    throwError! "invalid let declaration, term{indentExpr v}\nhas type{indentExpr vType}\nbut is expected to have type{indentExpr t}"
  | _ => unreachable!

private def checkConstant (constName : Name) (us : List Level) : MetaM Unit := do
  let cinfo ← getConstInfo constName
  unless us.length == cinfo.levelParams.length do
    throwIncorrectNumberOfLevels constName us

```

```lean
private def getFunctionDomain (f : Expr) : MetaM (Expr × BinderInfo) := do
  let fType ← inferType f
  let fType ← whnfD fType
  match fType with
  | Expr.forallE _ d _ c => return (d, c.binderInfo)
  | _                    => throwFunctionExpected f

/-
Given to expressions `a` and `b`, this method tries to annotate terms with `pp.explicit := true` to
expose "implicit" differences. For example, suppose `a` and `b` are of the form
```lean
@HashMap Nat Nat eqInst hasInst1
@HashMap Nat Nat eqInst hasInst2
```
By default, the pretty printer formats both of them as `HashMap Nat Nat`.
So, counterintuitive error messages such as
```lean
error: application type mismatch
  HashMap.insert m
argument
  m
has type
  HashMap Nat Nat
but is expected to have type
  HashMap Nat Nat
```
would be produced.
By adding `pp.explicit := true`, we can generate the more informative error
```lean
error: application type mismatch
  HashMap.insert m
argument
  m
has type
  @HashMap Nat Nat eqInst hasInst1
but is expected to have type
  @HashMap Nat Nat eqInst hasInst2
```
Remark: this method implements a simple heuristic, we should extend it as we find other counterintuitive
error messages.
-/
partial def addPPExplicitToExposeDiff (a b : Expr) : MetaM (Expr × Expr) := do
  if (← getOptions).getBool `pp.all false || (← getOptions).getBool `pp.explicit false then
    return (a, b)
  else
    visit a b
where
```

```
  visit (a b : Expr) : MetaM (Expr × Expr) := do
    try
      if !a.isApp || !b.isApp then
        return (a, b)
      else if a.getAppNumArgs != b.getAppNumArgs then
        return (a, b)
      else if not (← isDefEq a.getAppFn b.getAppFn) then
        return (a, b)
      else
        let fType ← inferType a.getAppFn
        forallBoundedTelescope fType a.getAppNumArgs fun xs _ => do
          let mut as := a.getAppArgs
          let mut bs := b.getAppArgs
          if (← hasExplicitDiff xs as bs) then
            return (a, b)
          else
            for i in [:as.size] do
              let (ai, bi) ← visit as[i] bs[i]
              as := as.set! i ai
              bs := bs.set! i bi
            let a := mkAppN a.getAppFn as
            let b := mkAppN b.getAppFn bs
            return (a.setAppPPExplicit, b.setAppPPExplicit)
    catch _ =>
      return (a, b)

  hasExplicitDiff (xs as bs : Array Expr) : MetaM Bool := do
    for i in [:xs.size] do
      let localDecl ← getLocalDecl xs[i].fvarId!
      if localDecl.binderInfo.isExplicit then
         if not (← isDefEq as[i] bs[i]) then
           return true
    return false

/-
  Return error message "has type{givenType}\nbut is expected to have type{expectedType}"
-/
def mkHasTypeButIsExpectedMsg (givenType expectedType : Expr) : MetaM MessageData := do
  let (givenType, expectedType) ← addPPExplicitToExposeDiff givenType expectedType
  m!"has type{indentExpr givenType}\nbut is expected to have type{indentExpr expectedType}"

def throwAppTypeMismatch {α} (f a : Expr) (extraMsg : MessageData := Format.nil) : MetaM α := do
  let (expectedType, binfo) ← getFunctionDomain f
  let mut e := mkApp f a
  unless binfo.isExplicit do
    e := e.setAppPPExplicit
  let aType ← inferType a
```

```
  throwError! "application type mismatch{indentExpr e}\nargument{indentExpr a}\n{← mkHasTypeButIsExpectedMsg aType expectedType}"

def checkApp (f a : Expr) : MetaM Unit := do
  let fType ← inferType f
  let fType ← whnf fType
  match fType with
  | Expr.forallE _ d _ _ =>
    let aType ← inferType a
    unless (← isDefEq d aType) do
      throwAppTypeMismatch f a
  | _ => throwFunctionExpected (mkApp f a)

private partial def checkAux : Expr → MetaM Unit
  | e@(Expr.forallE ..)  => checkForall e
  | e@(Expr.lam ..)      => checkLambdaLet e
  | e@(Expr.letE ..)     => checkLambdaLet e
  | Expr.const c lvls _  => checkConstant c lvls
  | Expr.app f a _       => do checkAux f; checkAux a; checkApp f a
  | Expr.mdata _ e _     => checkAux e
  | Expr.proj _ _ e _    => checkAux e
  | _                    => pure ()
where
  checkLambdaLet (e : Expr) : MetaM Unit :=
    lambdaLetTelescope e fun xs b => do
      xs.forM fun x => do
        let xDecl ← getFVarLocalDecl x;
        match xDecl with
        | LocalDecl.cdecl (type := t) .. =>
          ensureType t
          checkAux t
        | LocalDecl.ldecl (type := t) (value := v) .. =>
          ensureType t
          checkAux t
          let vType ← inferType v
          unless (← isDefEq t vType) do throwLetTypeMismatchMessage x.fvarId!
          checkAux v
      checkAux b

  checkForall (e : Expr) : MetaM Unit :=
    forallTelescope e fun xs b => do
      xs.forM fun x => do
        let xDecl ← getFVarLocalDecl x
        ensureType xDecl.type
        checkAux xDecl.type
      ensureType b
      checkAux b

```

```
def check (e : Expr) : MetaM Unit :=
  traceCtx `Meta.check do
    withTransparency TransparencyMode.all $ checkAux e

def isTypeCorrect (e : Expr) : MetaM Bool := do
  try
    check e
    pure true
  catch ex =>
    trace[Meta.typeError]! ex.toMessageData
    pure false

builtin_initialize
  registerTraceClass `Meta.check
  registerTraceClass `Meta.typeError

end Lean.Meta
// ::::::::::::::
// Closure.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Std.ShareCommon
import Lean.MetavarContext
import Lean.Environment
import Lean.Util.FoldConsts
import Lean.Meta.Basic
import Lean.Meta.Check

/-

This module provides functions for "closing" open terms and
creating auxiliary definitions. Here, we say a term is "open" if
it contains free/meta-variables.

The "closure" is performed by lambda abstracting the
free/meta-variables. Recall that in dependent type theory
lambda abstracting a let-variable may produce type incorrect terms.
For example, given the context
```lean
(n : Nat := 20)
(x : Vector α n)
(y : Vector α 20)
```
```

the term `x = y` is correct. However, its closure using lambda abstractions is not.

```lean
fun (n : Nat) (x : Vector α n) (y : Vector α 20) => x = y
```

A previous version of this module would address this issue by always use let-expressions to abstract let-vars. In the example above, it would produce

```lean
let n : Nat := 20; fun (x : Vector α n) (y : Vector α 20) => x = y
```

This approach produces correct result, but produces unsatisfactory results when we want to create auxiliary definitions. For example, consider the context

```lean
(x : Nat)
(y : Nat := fact x)
```

and the term `h (g y)`, now suppose we want to create an auxiliary definition for `y`. The previous version of this module would compute the auxiliary definition

```lean
def aux := fun (x : Nat) => let y : Nat := fact x; h (g y)
```

and would return the term `aux x` as a substitute for `h (g y)`. This is correct, but we will re-evaluate `fact x` whenever we use `aux`. In this module, we produce

```lean
def aux := fun (y : Nat) => h (g y)
```

Note that in this particular case, it is safe to lambda abstract the let-varible `y`. This module uses the following approach to decide whether it is safe or not to lambda abstract a let-variable.

1) We enable zeta-expansion tracking in `MetaM`. That is, whenever we perform type checking
   if a let-variable needs to zeta expanded, we store it in the set `zetaFVarIds`.
   We say a let-variable is zeta expanded when we replace it with its value.
2) We use the `MetaM` type checker `check` to type check the expression we want to close,
   and the type of the binders.
3) If a let-variable is not in `zetaFVarIds`, we lambda abstract it.

Remark: We still use let-expressions for let-variables in `zetaFVarIds`, but we move the `let` inside the lambdas. The idea is to make sure the auxiliary definition does not have an interleaving of `lambda` and `let` expressions. Thus, if the let-variable occurs in the type of one of the lambdas, we simply zeta-expand it there. As a final example consider the context

```lean
(x_1 : Nat)
(x_2 : Nat)
```

~~~~lean
(x_3 : Nat)
(x   : Nat := fact (10 + x_1 + x_2 + x_3))
(ty  : Type := Nat → Nat)
(f   : ty := fun x => x)
(n   : Nat := 20)
(z   : f 10)
```
and we use this module to compute an auxiliary definition for the term
```lean
(let y  : { v : Nat // v = n } := ⟨20, rfl⟩; y.1 + n + f x, z + 10)
```
we obtain
```lean
def aux (x : Nat) (f : Nat → Nat) (z : Nat) : Nat×Nat :=
let n : Nat := 20;
(let y : {v // v=n} := {val := 20, property := ex._proof_1}; y.val+n+f x, z+10)
```

BTW, this module also provides the `zeta : Bool` flag. When set to true, it
expands all let-variables occurring in the target expression.
-/

namespace Lean.Meta
namespace Closure

structure ToProcessElement where
  fvarId : FVarId
  newFVarId : FVarId
  deriving Inhabited

structure Context where
  zeta : Bool

structure State where
  visitedLevel          : LevelMap Level := {}
  visitedExpr           : ExprStructMap Expr := {}
  levelParams           : Array Name := #[]
  nextLevelIdx          : Nat := 1
  levelArgs             : Array Level := #[]
  newLocalDecls         : Array LocalDecl := #[]
  newLocalDeclsForMVars : Array LocalDecl := #[]
  newLetDecls           : Array LocalDecl := #[]
  nextExprIdx           : Nat := 1
  exprMVarArgs          : Array Expr := #[]
  exprFVarArgs          : Array Expr := #[]
  toProcess             : Array ToProcessElement := #[]

~~~~

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

@[inline] def visitLevel (f : Level → ClosureM Level) (u : Level) : ClosureM Level := do
  if !u.hasMVar && !u.hasParam then
    pure u
  else
    let s ← get
    match s.visitedLevel.find? u with
    | some v => pure v
    | none   => do
      let v ← f u
      modify fun s => { s with visitedLevel := s.visitedLevel.insert u v }
      pure v

@[inline] def visitExpr (f : Expr → ClosureM Expr) (e : Expr) : ClosureM Expr := do
  if !e.hasLevelParam && !e.hasFVar && !e.hasMVar then
    pure e
  else
    let s ← get
    match s.visitedExpr.find? e with
    | some r => pure r
    | none   =>
      let r ← f e
      modify fun s => { s with visitedExpr := s.visitedExpr.insert e r }
      pure r

def mkNewLevelParam (u : Level) : ClosureM Level := do
  let s ← get
  let p := (`u).appendIndexAfter s.nextLevelIdx
  modify fun s => { s with levelParams := s.levelParams.push p, nextLevelIdx := s.nextLevelIdx + 1, levelArgs := s.levelArgs.push u }
  pure $ mkLevelParam p

partial def collectLevelAux : Level → ClosureM Level
  | u@(Level.succ v _)      => return u.updateSucc! (← visitLevel collectLevelAux v)
  | u@(Level.max v w _)     => return u.updateMax! (← visitLevel collectLevelAux v) (← visitLevel collectLevelAux w)
  | u@(Level.imax v w _)    => return u.updateIMax! (← visitLevel collectLevelAux v) (← visitLevel collectLevelAux w)
  | u@(Level.mvar mvarId _) => mkNewLevelParam u
  | u@(Level.param _ _)     => mkNewLevelParam u
  | u@(Level.zero _)        => pure u

def collectLevel (u : Level) : ClosureM Level := do
  -- u ← instantiateLevelMVars u
  visitLevel collectLevelAux u

def preprocess (e : Expr) : ClosureM Expr := do
  let e ← instantiateMVars e
  let ctx ← read
```

```
  -- If we are not zeta-expanding let-decls, then we use `check` to find
  -- which let-decls are dependent. We say a let-decl is dependent if its lambda abstraction is type incorrect.
  if !ctx.zeta then
    check e
  pure e

/--
  Remark: This method does not guarantee unique user names.
  The correctness of the procedure does not rely on unique user names.
  Recall that the pretty printer takes care of unintended collisions. -/
def mkNextUserName : ClosureM Name := do
  let s ← get
  let n := (`_x).appendIndexAfter s.nextExprIdx
  modify fun s => { s with nextExprIdx := s.nextExprIdx + 1 }
  pure n

def pushToProcess (elem : ToProcessElement) : ClosureM Unit :=
  modify fun s => { s with toProcess := s.toProcess.push elem }

partial def collectExprAux (e : Expr) : ClosureM Expr := do
  let collect (e : Expr) := visitExpr collectExprAux e
  match e with
  | Expr.proj _ _ s _    => return e.updateProj! (← collect s)
  | Expr.forallE _ d b _ => return e.updateForallE! (← collect d) (← collect b)
  | Expr.lam _ d b _     => return e.updateLambdaE! (← collect d) (← collect b)
  | Expr.letE _ t v b _  => return e.updateLet! (← collect t) (← collect v) (← collect b)
  | Expr.app f a _       => return e.updateApp! (← collect f) (← collect a)
  | Expr.mdata _ b _     => return e.updateMData! (← collect b)
  | Expr.sort u _        => return e.updateSort! (← collectLevel u)
  | Expr.const c us _    => return e.updateConst! (← us.mapM collectLevel)
  | Expr.mvar mvarId _   =>
    let mvarDecl ← getMVarDecl mvarId
    let type ← preprocess mvarDecl.type
    let type ← collect type
    let newFVarId ← mkFreshFVarId
    let userName ← mkNextUserName
    modify fun s => { s with
      newLocalDeclsForMVars := s.newLocalDeclsForMVars.push $ LocalDecl.cdecl arbitrary newFVarId userName type BinderInfo.default,
      exprMVarArgs          := s.exprMVarArgs.push e
    }
    return mkFVar newFVarId
  | Expr.fvar fvarId _ =>
    match (← read).zeta, (← getLocalDecl fvarId).value? with
    | true, some value => collect (← preprocess value)
    | _,    _          =>
      let newFVarId ← mkFreshFVarId
      pushToProcess ⟨fvarId, newFVarId⟩
```

```
      return mkFVar newFVarId
  | e => pure e

def collectExpr (e : Expr) : ClosureM Expr := do
  let e ← preprocess e
  visitExpr collectExprAux e

partial def pickNextToProcessAux (lctx : LocalContext) (i : Nat) (toProcess : Array ToProcessElement) (elem : ToProcessElement)
    : ToProcessElement × Array ToProcessElement :=
  if h : i < toProcess.size then
    let elem' := toProcess.get ⟨i, h⟩
    if (lctx.get! elem.fvarId).index < (lctx.get! elem'.fvarId).index then
      pickNextToProcessAux lctx (i+1) (toProcess.set ⟨i, h⟩ elem) elem'
    else
      pickNextToProcessAux lctx (i+1) toProcess elem
  else
    (elem, toProcess)

def pickNextToProcess? : ClosureM (Option ToProcessElement) := do
  let lctx ← getLCtx
  let s ← get
  if s.toProcess.isEmpty then
    pure none
  else
    modifyGet fun s =>
      let elem      := s.toProcess.back
      let toProcess := s.toProcess.pop
      let (elem, toProcess) := pickNextToProcessAux lctx 0 toProcess elem
      (some elem, { s with toProcess := toProcess })

def pushFVarArg (e : Expr) : ClosureM Unit :=
  modify fun s => { s with exprFVarArgs := s.exprFVarArgs.push e }

def pushLocalDecl (newFVarId : FVarId) (userName : Name) (type : Expr) (bi := BinderInfo.default) : ClosureM Unit := do
  let type ← collectExpr type
  modify fun s => { s with newLocalDecls := s.newLocalDecls.push <| LocalDecl.cdecl arbitrary newFVarId userName type bi }

partial def process : ClosureM Unit := do
  match (← pickNextToProcess?) with
  | none => pure ()
  | some ⟨fvarId, newFVarId⟩ =>
    let localDecl ← getLocalDecl fvarId
    match localDecl with
    | LocalDecl.cdecl _ _ userName type bi =>
      pushLocalDecl newFVarId userName type bi
      pushFVarArg (mkFVar fvarId)
      process
```

```
    | LocalDecl.ldecl _ _ userName type val _ =>
      let zetaFVarIds ← getZetaFVarIds
      if !zetaFVarIds.contains fvarId then
        /- Non-dependent let-decl

            Recall that if `fvarId` is in `zetaFVarIds`, then we zeta-expanded it
            during type checking (see `check` at `collectExpr`).

            Our type checker may zeta-expand declarations that are not needed, but this
            check is conservative, and seems to work well in practice. -/
        pushLocalDecl newFVarId userName type
        pushFVarArg (mkFVar fvarId)
        process
      else
        /- Dependent let-decl -/
        let type ← collectExpr type
        let val  ← collectExpr val
        modify fun s => { s with newLetDecls := s.newLetDecls.push <| LocalDecl.ldecl arbitrary newFVarId userName type val false }
        /- We don't want to interleave let and lambda declarations in our closure. So, we expand any occurrences of newFVarId
           at `newLocalDecls` -/
        modify fun s => { s with newLocalDecls := s.newLocalDecls.map (replaceFVarIdAtLocalDecl newFVarId val) }
        process

@[inline] def mkBinding (isLambda : Bool) (decls : Array LocalDecl) (b : Expr) : Expr :=
  let xs := decls.map LocalDecl.toExpr
  let b  := b.abstract xs
  decls.size.foldRev (init := b) fun i b =>
    let decl := decls[i]
    match decl with
    | LocalDecl.cdecl _ _ n ty bi  =>
      let ty := ty.abstractRange i xs
      if isLambda then
        Lean.mkLambda n bi ty b
      else
        Lean.mkForall n bi ty b
    | LocalDecl.ldecl _ _ n ty val nonDep =>
      if b.hasLooseBVar 0 then
        let ty  := ty.abstractRange i xs
        let val := val.abstractRange i xs
        mkLet n ty val b nonDep
      else
        b.lowerLooseBVars 1 1

def mkLambda (decls : Array LocalDecl) (b : Expr) : Expr :=
  mkBinding true decls b

def mkForall (decls : Array LocalDecl) (b : Expr) : Expr :=
```

```
  mkBinding false decls b

structure MkValueTypeClosureResult where
  levelParams : Array Name
  type        : Expr
  value       : Expr
  levelArgs   : Array Level
  exprArgs    : Array Expr

def mkValueTypeClosureAux (type : Expr) (value : Expr) : ClosureM (Expr × Expr) := do
  resetZetaFVarIds
  withTrackingZeta do
    let type  ← collectExpr type
    let value ← collectExpr value
    process
    pure (type, value)

def mkValueTypeClosure (type : Expr) (value : Expr) (zeta : Bool) : MetaM MkValueTypeClosureResult := do
  let ((type, value), s) ← ((mkValueTypeClosureAux type value).run { zeta := zeta }).run {}
  let newLocalDecls := s.newLocalDecls.reverse ++ s.newLocalDeclsForMVars
  let newLetDecls   := s.newLetDecls.reverse
  let type  := mkForall newLocalDecls (mkForall newLetDecls type)
  let value := mkLambda newLocalDecls (mkLambda newLetDecls value)
  pure {
    type        := type,
    value       := value,
    levelParams := s.levelParams,
    levelArgs   := s.levelArgs,
    exprArgs    := s.exprFVarArgs.reverse ++ s.exprMVarArgs
  }

end Closure

/--
  Create an auxiliary definition with the given name, type and value.
  The parameters `type` and `value` may contain free and meta variables.
  A "closure" is computed, and a term of the form `name.{u_1 ... u_n} t_1 ... t_m` is
  returned where `u_i`s are universe parameters and metavariables `type` and `value` depend on,
  and `t_j`s are free and meta variables `type` and `value` depend on. -/
def mkAuxDefinition (name : Name) (type : Expr) (value : Expr) (zeta : Bool := false) (compile : Bool := true) : MetaM Expr := do
  trace[Meta.debug]! "{name} : {type} := {value}"
  let result ← Closure.mkValueTypeClosure type value zeta
  let env ← getEnv
  let decl := Declaration.defnDecl {
    name        := name,
    levelParams := result.levelParams.toList,
    type        := result.type,
```

```
    value       := result.value,
    hints       := ReducibilityHints.regular (getMaxHeight env result.value + 1),
    safety      := if env.hasUnsafe result.type || env.hasUnsafe result.value then DefinitionSafety.unsafe else DefinitionSafety.safe
  }
  trace[Meta.debug]! "{name} : {result.type} := {result.value}"
  addDecl decl
  if compile then
    compileDecl decl
  return mkAppN (mkConst name result.levelArgs.toList) result.exprArgs

/-- Similar to `mkAuxDefinition`, but infers the type of `value`. -/
def mkAuxDefinitionFor (name : Name) (value : Expr) : MetaM Expr := do
  let type ← inferType value
  let type := type.headBeta
  mkAuxDefinition name type value

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

namespace Lean.Meta

/--
  Return true iff `declName` is one of the auxiliary definitions/projections
  used to implement coercions.
-/
def isCoeDecl (declName : Name) : Bool :=
  declName == ``coe  ||
  declName == ``coeB ||  declName == ``coeHead || declName == ``coeTail || declName == ``coeD ||
  declName == ``coeTC || declName == ``coeFun || declName == ``coeSort ||
  declName == ``Coe.coe || declName == ``CoeTC.coe || declName == ``CoeHead.coe ||
  declName == ``CoeTail.coe || declName == ``CoeHTCT.coe || declName == ``CoeDep.coe ||
  declName == ``CoeT.coe || declName == ``CoeFun.coe || declName == ``CoeSort.coe ||
  declName == ``liftCoeM || declName == ``coeM

/-- Expand coercions occurring in `e` -/
partial def expandCoe (e : Expr) : MetaM Expr :=
  withReducibleAndInstances do
    return (← transform e (pre := step))
```

```
where
  step (e : Expr) : MetaM TransformStep := do
    let f := e.getAppFn
    if !f.isConst then
      return TransformStep.visit e
    else
      let declName := f.constName!
      if isCoeDecl declName then
        match (← unfoldDefinition? e) with
        | none   => return TransformStep.visit e
        | some e => step e.headBeta
      else
        return TransformStep.visit e

end Lean.Meta
// ::::::::::::::
// CollectMVars.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.CollectMVars
import Lean.Meta.Basic

namespace Lean.Meta

/--
  Collect unassigned metavariables occuring in the given expression.

  Remark: if `e` contains `?m` and there is a `t` assigned to `?m`, we
  collect unassigned metavariables occurring in `t`.

  Remark: if `e` contains `?m` and `?m` is delayed assigned to some term `t`,
  we collect `?m` and unassigned metavariables occurring in `t`.
  We collect `?m` because it has not been assigned yet. -/
partial def collectMVars (e : Expr) : StateRefT CollectMVars.State MetaM Unit := do
  let e ← instantiateMVars e
  let s ← get
  let resultSavedSize := s.result.size
  let s := e.collectMVars s
  set s
  for mvarId in s.result[resultSavedSize:] do
    match (← getDelayedAssignment? mvarId) with
    | none   => pure ()
    | some d => collectMVars d.val
```

```
/-- Return metavariables in occuring the given expression. See `collectMVars` -/
def getMVars (e : Expr) : MetaM (Array MVarId) := do
  let (_, s) ← (collectMVars e).run {}
  pure s.result

/-- Similar to getMVars, but removes delayed assignments. -/
def getMVarsNoDelayed (e : Expr) : MetaM (Array MVarId) := do
  let mvarIds ← getMVars e
  mvarIds.filterM fun mvarId => not <$> isDelayedAssigned mvarId

def collectMVarsAtDecl (d : Declaration) : StateRefT CollectMVars.State MetaM Unit :=
  d.forExprM collectMVars

def getMVarsAtDecl (d : Declaration) : MetaM (Array MVarId) := do
  let (_, s) ← (collectMVarsAtDecl d).run {}
  pure s.result

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

namespace Lean.Meta.DiscrTree
/-
  (Imperfect) discrimination trees.
  We use a hybrid representation.
  - A `PersistentHashMap` for the root node which usually contains many children.
  - A sorted array of key/node pairs for inner nodes.

  The edges are labeled by keys:
  - Constant names (and arity). Universe levels are ignored.
  - Free variables (and arity). Thus, an entry in the discrimination tree
    may reference hypotheses from the local context.
  - Literals
  - Star/Wildcard. We use them to represent metavariables and terms
    we want to ignore. We ignore implicit arguments and proofs.
  - Other. We use to represent other kinds of terms (e.g., nested lambda, forall, sort, etc).

```

```
  We reduce terms using `TransparencyMode.reducible`. Thus, all reducible
  definitions in an expression `e` are unfolded before we insert it into the
  discrimination tree.

  Recall that projections from classes are **NOT** reducible.
  For example, the expressions `Add.add α (ringAdd ?α ?s) ?x ?x`
  and `Add.add Nat Nat.hasAdd a b` generates paths with the following keys
  respctively
  ```
  ⟨Add.add, 4⟩, *, *, *, *
  ⟨Add.add, 4⟩, *, *, ⟨a,0⟩, ⟨b,0⟩
  ```

  That is, we don't reduce `Add.add Nat inst a b` into `Nat.add a b`.
  We say the `Add.add` applications are the de-facto canonical forms in
  the metaprogramming framework.
  Moreover, it is the metaprogrammer's responsibility to re-pack applications such as
  `Nat.add a b` into `Add.add Nat inst a b`.

  Remark: we store the arity in the keys
  1- To be able to implement the "skip" operation when retrieving "candidate"
     unifiers.
  2- Distinguish partial applications `f a`, `f a b`, and `f a b c`.
-/

def Key.ctorIdx : Key → Nat
  | Key.star   => 0
  | Key.other   => 1
  | Key.lit _   => 2
  | Key.fvar _ _ => 3
  | Key.const _ _ => 4

def Key.lt : Key → Key → Bool
  | Key.lit v₁,   Key.lit v₂   => v₁ < v₂
  | Key.fvar n₁ a₁, Key.fvar n₂ a₂ => Name.quickLt n₁ n₂ || (n₁ == n₂ && a₁ < a₂)
  | Key.const n₁ a₁, Key.const n₂ a₂ => Name.quickLt n₁ n₂ || (n₁ == n₂ && a₁ < a₂)
  | k₁,   k₂   => k₁.ctorIdx < k₂.ctorIdx

instance : HasLess Key := ⟨fun a b => Key.lt a b⟩
instance (a b : Key) : Decidable (a < b) := inferInstanceAs (Decidable (Key.lt a b))

def Key.format : Key → Format
  | Key.star   => "*"
  | Key.other   => "■"
  | Key.lit (Literal.natVal v) => fmt v
  | Key.lit (Literal.strVal v) => repr v
  | Key.const k _   => fmt k
```

```
  | Key.fvar k _   => fmt k

instance : ToFormat Key := ⟨Key.format⟩

def Key.arity : Key → Nat
  | Key.const _ a => a
  | Key.fvar _ a => a
  | _   => 0

instance {α} : Inhabited (Trie α) := ⟨Trie.node #[] #[]⟩

def empty {α} : DiscrTree α := { root := {} }

partial def Trie.format {α} [ToFormat α] : Trie α → Format
  | Trie.node vs cs => Format.group $ Format.paren $
    "node" ++ (if vs.isEmpty then Format.nil else " " ++ fmt vs)
    ++ Format.join (cs.toList.map $ fun ⟨k, c⟩ => Format.line ++ Format.paren (fmt k ++ " => " ++ format c))

instance {α} [ToFormat α] : ToFormat (Trie α) := ⟨Trie.format⟩

partial def format {α} [ToFormat α] (d : DiscrTree α) : Format :=
  let (_, r) := d.root.foldl
    (fun (p : Bool × Format) k c =>
      (false, p.2 ++ (if p.1 then Format.nil else Format.line) ++ Format.paren (fmt k ++ " => " ++ fmt c)))
    (true, Format.nil)
  Format.group r

instance {α} [ToFormat α] : ToFormat (DiscrTree α) := ⟨format⟩

/- The discrimination tree ignores implicit arguments and proofs.
   We use the following auxiliary id as a "mark". -/
private def tmpMVarId : MVarId := `_discr_tree_tmp
private def tmpStar := mkMVar tmpMVarId

instance {α} : Inhabited (DiscrTree α) where
  default := {}

/--
  Return true iff the argument should be treated as a "wildcard" by the discrimination tree.

  - We ignore proofs because of proof irrelevance. It doesn't make sense to try to
    index their structure.

  - We ignore instance implicit arguments (e.g., `[Add α]`) because they are "morally" canonical.
    Moreover, we may have many definitionally equal terms floating around.
    Example: `Ring.hasAdd Int Int.isRing` and `Int.hasAdd`.

```

```
  - We considered ignoring implicit arguments (e.g., `{α : Type}`) since users don't "see" them,
    and may not even understand why some simplification rule is not firing.
    However, in type class resolution, we have instance such as `Decidable (@Eq Nat x y)`,
    where `Nat` is an implicit argument. Thus, we would add the path
    ```
    Decidable -> Eq -> * -> * -> * -> [Nat.decEq]
    ```
    to the discrimination tree IF we ignored the implict `Nat` argument.
    This would be BAD since **ALL** decidable equality instances would be in the same path.
    So, we index implicit arguments if they are types.
    This setting seems sensible for simplification lemmas such as:
    ```
    forall (x y : Unit), (@Eq Unit x y) = true
    ```
    If we ignore the implicit argument `Unit`, the `DiscrTree` will say it is a candidate
    simplification lemma for any equality in our goal.

  Remark: if users have problems with the solution above, we may provide a `noIndexing` annotation,
  and `ignoreArg` would return true for any term of the form `noIndexing t`.
-/
private def ignoreArg (a : Expr) (i : Nat) (infos : Array ParamInfo) : MetaM Bool :=
  if h : i < infos.size then
    let info := infos.get ⟨i, h⟩
    if info.instImplicit then
      pure true
    else if info.implicit then
      not <$> isType a
    else
      isProof a
  else
    isProof a

private partial def pushArgsAux (infos : Array ParamInfo) : Nat → Expr → Array Expr → MetaM (Array Expr)
  | i, Expr.app f a _, todo => do
    if (← ignoreArg a i infos) then
      pushArgsAux infos (i-1) f (todo.push tmpStar)
    else
      pushArgsAux infos (i-1) f (todo.push a)
  | _, _, todo => pure todo

private partial def whnfEta (e : Expr) : MetaM Expr := do
  let e ← whnf e
  match e.etaExpandedStrict? with
  | some e => whnfEta e
  | none   => pure e

/-
```

```
  TODO: add a parameter (wildcardConsts : NameSet) to `DiscrTree.insert`.
  Then, `DiscrTree` users may control which symbols should be treated as wildcards.
  Different `DiscrTree` users may populate this set using, for example, attributes.

  Remark: we currently tag `Nat.zero` and `Nat.succ` to avoid having to add special
  support for `Expr.lit`. Example, suppose the discrimination tree contains the entry
  `Nat.succ ?m |-> v`, and we are trying to retrieve the matches for `Expr.lit (Literal.natVal 1) _`.
  In this scenario, we want to retrieve `Nat.succ ?m |-> v` -/
private def shouldAddAsStar (constName : Name) : Bool :=
  constName == `Nat.zero || constName == `Nat.succ

def mkNoindexAnnotation (e : Expr) : Expr :=
  mkAnnotation `noindex e

def hasNoindexAnnotation (e : Expr) : Bool :=
  annotation? `noindex e |>.isSome

/- Remark: we use `shouldAddAsStar` only for nested terms, and `first == false` for nested terms -/

private def pushArgs (first : Bool) (todo : Array Expr) (e : Expr) : MetaM (Key × Array Expr) := do
  if hasNoindexAnnotation e then
    return (Key.star, todo)
  else
    let e ← whnfEta e
    let fn := e.getAppFn
    let push (k : Key) (nargs : Nat) : MetaM (Key × Array Expr) := do
      let info ← getFunInfoNArgs fn nargs
      let todo ← pushArgsAux info.paramInfo (nargs-1) e todo
      return (k, todo)
    match fn with
    | Expr.lit v _       => return (Key.lit v, todo)
    | Expr.const c _ _   =>
      if !first && shouldAddAsStar c then
        return (Key.star, todo)
      else
        let nargs := e.getAppNumArgs
        push (Key.const c nargs) nargs
    | Expr.fvar fvarId _ =>
      let nargs := e.getAppNumArgs
      push (Key.fvar fvarId nargs) nargs
    | Expr.mvar mvarId _ =>
      if mvarId == tmpMVarId then
        -- We use `tmp to mark implicit arguments and proofs
        return (Key.star, todo)
      else if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
        return (Key.other, todo)
      else
```

```
        return (Key.star, todo)
    | _ =>
      return (Key.other, todo)

partial def mkPathAux (first : Bool) (todo : Array Expr) (keys : Array Key) : MetaM (Array Key) := do
  if todo.isEmpty then
    pure keys
  else
    let e    := todo.back
    let todo := todo.pop
    let (k, todo) ← pushArgs first todo e
    mkPathAux false todo (keys.push k)

private def initCapacity := 8

def mkPath (e : Expr) : MetaM (Array Key) := do
  withReducible do
    let todo : Array Expr := Array.mkEmpty initCapacity
    let keys : Array Key  := Array.mkEmpty initCapacity
    mkPathAux (first := true) (todo.push e) keys

private partial def createNodes {α} (keys : Array Key) (v : α) (i : Nat) : Trie α :=
  if h : i < keys.size then
    let k := keys.get ⟨i, h⟩
    let c := createNodes keys v (i+1)
    Trie.node #[] #[(k, c)]
  else
    Trie.node #[v] #[]

private def insertVal {α} [BEq α] (vs : Array α) (v : α) : Array α :=
  if vs.contains v then vs else vs.push v

private partial def insertAux {α} [BEq α] (keys : Array Key) (v : α) : Nat → Trie α → Trie α
  | i, Trie.node vs cs =>
    if h : i < keys.size then
      let k := keys.get ⟨i, h⟩
      let c := Id.run $ cs.binInsertM
          (fun a b => a.1 < b.1)
          (fun ⟨_, s⟩ => let c := insertAux keys v (i+1) s; (k, c)) -- merge with existing
          (fun _ => let c := createNodes keys v (i+1); (k, c))
          (k, arbitrary)
      Trie.node vs c
    else
      Trie.node (insertVal vs v) cs

def insertCore {α} [BEq α] (d : DiscrTree α) (keys : Array Key) (v : α) : DiscrTree α :=
  if keys.isEmpty then panic! "invalid key sequence"
```

```
  else
    let k := keys[0]
    match d.root.find? k with
    | none =>
      let c := createNodes keys v 1
      { root := d.root.insert k c }
    | some c =>
      let c := insertAux keys v 1 c
      { root := d.root.insert k c }

def insert {α} [BEq α] (d : DiscrTree α) (e : Expr) (v : α) : MetaM (DiscrTree α) := do
  let keys ← mkPath e
  return d.insertCore keys v

private def getKeyArgs (e : Expr) (isMatch? : Bool) : MetaM (Key × Array Expr) := do
  let e ← whnfEta e
  match e.getAppFn with
  | Expr.lit v _       => pure (Key.lit v, #[])
  | Expr.const c _ _   =>
    let nargs := e.getAppNumArgs
    pure (Key.const c nargs, e.getAppRevArgs)
  | Expr.fvar fvarId _ =>
    let nargs := e.getAppNumArgs
    pure (Key.fvar fvarId nargs, e.getAppRevArgs)
  | Expr.mvar mvarId _ =>
    if isMatch? then
      pure (Key.other, #[])
    else do
      let ctx ← read
      if ctx.config.isDefEqStuckEx then
        /-
          When the configuration flag `isDefEqStuckEx` is set to true,
          we want `isDefEq` to throw an exception whenever it tries to assign
          a read-only metavariable.
          This feature is useful for type class resolution where
          we may want to notify the caller that the TC problem may be solveable
          later after it assigns `?m`.
          The method `DiscrTree.getUnify e` returns candidates `c` that may "unify" with `e`.
          That is, `isDefEq c e` may return true. Now, consider `DiscrTree.getUnify d (Add ?m)`
          where `?m` is a read-only metavariable, and the discrimination tree contains the keys
          `HadAdd Nat` and `Add Int`. If `isDefEqStuckEx` is set to true, we must treat `?m` as
          a regular metavariable here, otherwise we return the empty set of candidates.
          This is incorrect because it is equivalent to saying that there is no solution even if
          the caller assigns `?m` and try again. -/
        pure (Key.star, #[])
      else if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
        pure (Key.other, #[])
```

```
      else
        pure (Key.star, #[])
  | _ => pure (Key.other, #[])

private abbrev getMatchKeyArgs (e : Expr) : MetaM (Key × Array Expr) :=
  getKeyArgs e true

private abbrev getUnifyKeyArgs (e : Expr) : MetaM (Key × Array Expr) :=
  getKeyArgs e false

private def getStarResult {α} (d : DiscrTree α) : Array α :=
  let result : Array α := Array.mkEmpty initCapacity
  match d.root.find? Key.star with
  | none                  => result
  | some (Trie.node vs _) => result ++ vs

partial def getMatch {α} (d : DiscrTree α) (e : Expr) : MetaM (Array α) :=
  withReducible do
    let result := getStarResult d
    let (k, args) ← getMatchKeyArgs e
    match k with
    | Key.star => pure result
    | _        =>
      match d.root.find? k with
      | none   => pure result
      | some c => process args c result
where
  process (todo : Array Expr) (c : Trie α) (result : Array α) : MetaM (Array α) := do
    match c with
    | Trie.node vs cs =>
      if todo.isEmpty then
        return result ++ vs
      else if cs.isEmpty then
        return result
      else
        let e     := todo.back
        let todo  := todo.pop
        let first := cs[0] /- Recall that `Key.star` is the minimal key -/
        let (k, args) ← getMatchKeyArgs e
        /- We must always visit `Key.star` edges since they are wildcards.
           Thus, `todo` is not used linearly when there is `Key.star` edge
           and there is an edge for `k` and `k != Key.star`. -/
        let visitStarChild (result : Array α) : MetaM (Array α) :=
          if first.1 == Key.star then
            process todo first.2 result
          else
            return result
```

```
        match k with
        | Key.star => visitStarChild result
        | _ =>
          match cs.binSearch (k, arbitrary) (fun a b => a.1 < b.1) with
          | none   => visitStarChild result
          | some c =>
            let result ← visitStarChild result
            process (todo ++ args) c.2 result

partial def getUnify {α} (d : DiscrTree α) (e : Expr) : MetaM (Array α) :=
  withReducible do
    let (k, args) ← getUnifyKeyArgs e
    match k with
    | Key.star => d.root.foldlM (init := #[]) fun result k c => process k.arity #[] c result
    | _ =>
      let result := getStarResult d
      match d.root.find? k with
      | none   => return result
      | some c => process 0 args c result
where
  process (skip : Nat) (todo : Array Expr) (c : Trie α) (result : Array α) : MetaM (Array α) := do
    match skip, c with
    | skip+1, Trie.node vs cs =>
      if cs.isEmpty then
        return result
      else
        cs.foldlM (init := result) fun result ⟨k, c⟩ => process (skip + k.arity) todo c result
    | 0, Trie.node vs cs => do
      if todo.isEmpty then
        return result ++ vs
      else if cs.isEmpty then
        return result
      else
        let e     := todo.back
        let todo  := todo.pop
        let (k, args) ← getUnifyKeyArgs e
        match k with
        | Key.star => cs.foldlM (init := result) fun result ⟨k, c⟩ => process k.arity todo c result
        | _ =>
          let first := cs[0]
          let visitStarChild (result : Array α) : MetaM (Array α) :=
            if first.1 == Key.star then
              process 0 todo first.2 result
            else
              return result
          match cs.binSearch (k, arbitrary) (fun a b => a.1 < b.1) with
          | none   => visitStarChild result
```

```
          | some c => process 0 (todo ++ args) c.2 (← visitStarChild result)

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

namespace Lean.Meta

/- See file `DiscrTree.lean` for the actual implementation and documentation. -/

namespace DiscrTree

inductive Key where
  | const : Name → Nat → Key
  | fvar  : FVarId → Nat → Key
  | lit   : Literal → Key
  | star  : Key
  | other : Key
  deriving Inhabited, BEq

protected def Key.hash : Key → USize
  | Key.const n a => mixHash 5237 $ mixHash (hash n) (hash a)
  | Key.fvar n a  => mixHash 3541 $ mixHash (hash n) (hash a)
  | Key.lit v     => mixHash 1879 $ hash v
  | Key.star      => 7883
  | Key.other     => 2411

instance : Hashable Key := ⟨Key.hash⟩

inductive Trie (α : Type) where
  | node (vs : Array α) (children : Array (Key × Trie α)) : Trie α

end DiscrTree

open DiscrTree
open Std (PersistentHashMap)

structure DiscrTree (α : Type) where
  root : PersistentHashMap Key (Trie α) := {}

```

```
end Lean.Meta
// ::::::::::::::
// ExprDefEq.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.ProjFns
import Lean.Meta.WHNF
import Lean.Meta.InferType
import Lean.Meta.FunInfo
import Lean.Meta.LevelDefEq
import Lean.Meta.Check
import Lean.Meta.Offset
import Lean.Meta.ForEachExpr
import Lean.Meta.UnificationHint

namespace Lean.Meta

/--
  Try to solve `a := (fun x => t) =?= b` by eta-expanding `b`.

  Remark: eta-reduction is not a good alternative even in a system without universe cumulativity like Lean.
  Example:
    ```
    (fun x : A => f ?m) =?= f
    ```
    The left-hand side of the constraint above it not eta-reduced because `?m` is a metavariable. -/
private def isDefEqEta (a b : Expr) : MetaM Bool := do
  if a.isLambda && !b.isLambda then
    let bType ← inferType b
    let bType ← whnfD bType
    match bType with
    | Expr.forallE n d _ c =>
      let b' := mkLambda n c.binderInfo d (mkApp b (mkBVar 0))
      commitWhen <| Meta.isExprDefEqAux a b'
    | _ => pure false
  else
    pure false

/-- Support for `Lean.reduceBool` and `Lean.reduceNat` -/
def isDefEqNative (s t : Expr) : MetaM LBool := do
  let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
  let s? ← reduceNative? s
  let t? ← reduceNative? t
```

```
  match s?, t? with
  | some s, some t => isDefEq s t
  | some s, none   => isDefEq s t
  | none,   some t => isDefEq s t
  | none,   none   => pure LBool.undef

/-- Support for reducing Nat basic operations. -/
def isDefEqNat (s t : Expr) : MetaM LBool := do
  let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
  if s.hasFVar || s.hasMVar || t.hasFVar || t.hasMVar then
    pure LBool.undef
  else
    let s? ← reduceNat? s
    let t? ← reduceNat? t
    match s?, t? with
    | some s, some t => isDefEq s t
    | some s, none   => isDefEq s t
    | none,   some t => isDefEq s t
    | none,   none   => pure LBool.undef

/-- Support for constraints of the form `("..." =?= String.mk cs)` -/
def isDefEqStringLit (s t : Expr) : MetaM LBool := do
  let isDefEq (s t) : MetaM LBool := toLBoolM <| Meta.isExprDefEqAux s t
  if s.isStringLit && t.isAppOf `String.mk then
    isDefEq (toCtorIfLit s) t
  else if s.isAppOf `String.mk && t.isStringLit then
    isDefEq s (toCtorIfLit t)
  else
    pure LBool.undef

/--
  Return `true` if `e` is of the form `fun (x_1 ... x_n) => ?m x_1 ... x_n)`, and `?m` is unassigned.
  Remark: `n` may be 0. -/
def isEtaUnassignedMVar (e : Expr) : MetaM Bool := do
  match e.etaExpanded? with
  | some (Expr.mvar mvarId _) =>
    if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
      pure false
    else if (← isExprMVarAssigned mvarId) then
      pure false
    else
      pure true
  | _   => pure false

/-
  First pass for `isDefEqArgs`. We unify explicit arguments, *and* easy cases
  Here, we say a case is easy if it is of the form
```

```
       ?m =?= t
       or
       t  =?= ?m

  where `?m` is unassigned.

  These easy cases are not just an optimization. When
  `?m` is a function, by assigning it to t, we make sure
  a unification constraint (in the explicit part)
  ```
  ?m t =?= f s
  ```
  is not higher-order.

  We also handle the eta-expanded cases:
  ```
  fun x₁ ... xₙ => ?m x₁ ... xₙ =?= t
  t =?= fun x₁ ... xₙ => ?m x₁ ... xₙ

  This is important because type inference often produces
  eta-expanded terms, and without this extra case, we could
  introduce counter intuitive behavior.

  Pre: `paramInfo.size <= args₁.size = args₂.size`
-/
private partial def isDefEqArgsFirstPass
    (paramInfo : Array ParamInfo) (args₁ args₂ : Array Expr) : MetaM (Option (Array Nat)) := do
  let rec loop (i : Nat) (postponed : Array Nat) := do
    if h : i < paramInfo.size then
      let info := paramInfo.get ⟨i, h⟩
      let a₁ := args₁[i]
      let a₂ := args₂[i]
      if info.implicit || info.instImplicit then
        if (← isEtaUnassignedMVar a₁ <||> isEtaUnassignedMVar a₂) then
          if (← Meta.isExprDefEqAux a₁ a₂) then
            loop (i+1) postponed
          else
            pure none
        else
          loop (i+1) (postponed.push i)
      else if (← Meta.isExprDefEqAux a₁ a₂) then
        loop (i+1) postponed
      else
        pure none
    else
      pure (some postponed)
```
```

```
  loop 0 #[]

@[specialize] private def trySynthPending (e : Expr) : MetaM Bool := do
  let mvarId? ← getStuckMVar? e
  match mvarId? with
  | some mvarId => Meta.synthPending mvarId
  | none        => pure false

private partial def isDefEqArgs (f : Expr) (args₁ args₂ : Array Expr) : MetaM Bool :=
  if h : args₁.size = args₂.size then do
    let finfo ← getFunInfoNArgs f args₁.size
    let (some postponed) ← isDefEqArgsFirstPass finfo.paramInfo args₁ args₂ | pure false
    let rec processOtherArgs (i : Nat) : MetaM Bool := do
      if h₁ : i < args₁.size then
        let a₁ := args₁.get ⟨i, h₁⟩
        let a₂ := args₂.get ⟨i, Eq.subst h h₁⟩
        if (← Meta.isExprDefEqAux a₁ a₂) then
          processOtherArgs (i+1)
        else
          pure false
      else
        pure true
    if (← processOtherArgs finfo.paramInfo.size) then
      postponed.allM fun i => do
        /- Second pass: unify implicit arguments.
           In the second pass, we make sure we are unfolding at
           least non reducible definitions (default setting). -/
        let a₁   := args₁[i]
        let a₂   := args₂[i]
        let info := finfo.paramInfo[i]
        if info.instImplicit then
          discard <| trySynthPending a₁
          discard <| trySynthPending a₂
        withAtLeastTransparency TransparencyMode.default <| Meta.isExprDefEqAux a₁ a₂
    else
      pure false
  else
    pure false

/--
  Check whether the types of the free variables at `fvars` are
  definitionally equal to the types at `ds₂`.

  Pre: `fvars.size == ds₂.size`

  This method also updates the set of local instances, and invokes
  the continuation `k` with the updated set.
```

```

  We can't use `withNewLocalInstances` because the `isDeq fvarType d₂`
  may use local instances. -/
@[specialize] partial def isDefEqBindingDomain (fvars : Array Expr) (ds₂ : Array Expr) (k : MetaM Bool) : MetaM Bool :=
  let rec loop (i : Nat) := do
    if h : i < fvars.size then do
      let fvar := fvars.get ⟨i, h⟩
      let fvarDecl ← getFVarLocalDecl fvar
      let fvarType := fvarDecl.type
      let d₂       := ds₂[i]
      if (← Meta.isExprDefEqAux fvarType d₂) then
        match (← isClass? fvarType) with
        | some className => withNewLocalInstance className fvar <| loop (i+1)
        | none           => loop (i+1)
      else
        pure false
    else
      k
  loop 0

/- Auxiliary function for `isDefEqBinding` for handling binders `forall/fun`.
   It accumulates the new free variables in `fvars`, and declare them at `lctx`.
   We use the domain types of `e₁` to create the new free variables.
   We store the domain types of `e₂` at `ds₂`. -/
private partial def isDefEqBindingAux (lctx : LocalContext) (fvars : Array Expr) (e₁ e₂ : Expr) (ds₂ : Array Expr) : MetaM Bool :=
  let process (n : Name) (d₁ d₂ b₁ b₂ : Expr) : MetaM Bool := do
    let d₁     := d₁.instantiateRev fvars
    let d₂     := d₂.instantiateRev fvars
    let fvarId ← mkFreshId
    let lctx   := lctx.mkLocalDecl fvarId n d₁
    let fvars  := fvars.push (mkFVar fvarId)
    isDefEqBindingAux lctx fvars b₁ b₂ (ds₂.push d₂)
  match e₁, e₂ with
  | Expr.forallE n d₁ b₁ _, Expr.forallE _ d₂ b₂ _ => process n d₁ d₂ b₁ b₂
  | Expr.lam     n d₁ b₁ _, Expr.lam     _ d₂ b₂ _ => process n d₁ d₂ b₁ b₂
  | _,                      _                      =>
    withReader (fun ctx => { ctx with lctx := lctx }) do
      isDefEqBindingDomain fvars ds₂ do
        Meta.isExprDefEqAux (e₁.instantiateRev fvars) (e₂.instantiateRev fvars)

@[inline] private def isDefEqBinding (a b : Expr) : MetaM Bool := do
  let lctx ← getLCtx
  isDefEqBindingAux lctx #[] a b #[]

private def checkTypesAndAssign (mvar : Expr) (v : Expr) : MetaM Bool :=
  traceCtx `Meta.isDefEq.assign.checkTypes do
    if !mvar.isMVar then
```

```
      trace[Meta.isDefEq.assign.final]! "metavariable expected at {mvar} := {v}"
      return false
    else
      -- must check whether types are definitionally equal or not, before assigning and returning true
      let mvarType ← inferType mvar
      let vType ← inferType v
      if (← withTransparency TransparencyMode.default <| Meta.isExprDefEqAux mvarType vType) then
        trace[Meta.isDefEq.assign.final]! "{mvar} := {v}"
        assignExprMVar mvar.mvarId! v
        pure true
      else
        trace[Meta.isDefEq.assign.typeMismatch]! "{mvar} : {mvarType} := {v} : {vType}"
        pure false

/--
  Auxiliary method for solving constraints of the form `?m xs := v`.
  It creates a lambda using `mkLambdaFVars ys v`, where `ys` is a superset of `xs`.
  `ys` is often equal to `xs`. It is a bigger when there are let-declaration dependencies in `xs`.
  For example, suppose we have `xs` of the form `#[a, c]` where
  ```
  a : Nat
  b : Nat := f a
  c : b = a
  ```
  In this scenario, the type of `?m` is `(x1 : Nat) -> (x2 : f x1 = x1) -> C[x1, x2]`,
  and type of `v` is `C[a, c]`. Note that, `?m a c` is type correct since `f a = a` is definitionally equal
  to the type of `c : b = a`, and the type of `?m a c` is equal to the type of `v`.
  Note that `fun xs => v` is the term `fun (x1 : Nat) (x2 : b = x1) => v` which has type
  `(x1 : Nat) -> (x2 : b = x1) -> C[x1, x2]` which is not definitionally equal to the type of `?m`,
  and may not even be type correct.
  The issue here is that we are not capturing the `let`-declarations.

  This method collects let-declarations `y` occurring between `xs[0]` and `xs.back` s.t.
  some `x` in `xs` depends on `y`.
  `ys` is the `xs` with these extra let-declarations included.

  In the example above, `ys` is `#[a, b, c]`, and `mkLambdaFVars ys v` produces
  `fun a => let b := f a; fun (c : b = a) => v` which has a type definitionally equal to the type of `?m`.

  Recall that the method `checkAssignment` ensures `v` does not contain offending `let`-declarations.

  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]`.
  where the index is the position in the local context.
-/
private partial def mkLambdaFVarsWithLetDeps (xs : Array Expr) (v : Expr) : MetaM (Option Expr) := do
  if not (← hasLetDeclsInBetween) then
    mkLambdaFVars xs v
```

```
  else
    let ys ← addLetDeps
    trace[Meta.debug]! "ys: {ys}, v: {v}"
    mkLambdaFVars ys v

where
  /- Return true if there are let-declarions between `xs[0]` and `xs[xs.size-1]`.
     We use it a quick-check to avoid the more expensive collection procedure. -/
  hasLetDeclsInBetween : MetaM Bool := do
    let check (lctx : LocalContext) : Bool := do
      let start := lctx.getFVar! xs[0] |>.index
      let stop  := lctx.getFVar! xs.back |>.index
      for i in [start+1:stop] do
        match lctx.getAt! i with
        | some localDecl =>
          if localDecl.isLet then
            return true
        | _ => pure ()
      return false
    if xs.size <= 1 then
      pure false
    else
      check (← getLCtx)

  /- Traverse `e` and stores in the state `NameHashSet` any let-declaration with index greater than `(← read)`.
     The context `Nat` is the position of `xs[0]` in the local context. -/
  collectLetDeclsFrom (e : Expr) : ReaderT Nat (StateRefT NameHashSet MetaM) Unit := do
    let rec visit (e : Expr) : MonadCacheT Expr Unit (ReaderT Nat (StateRefT NameHashSet MetaM)) Unit :=
      checkCache e fun _ => do
        match e with
        | Expr.forallE _ d b _   => visit d; visit b
        | Expr.lam _ d b _       => visit d; visit b
        | Expr.letE _ t v b _    => visit t; visit v; visit b
        | Expr.app f a _         => visit f; visit a
        | Expr.mdata _ b _       => visit b
        | Expr.proj _ _ b _      => visit b
        | Expr.fvar fvarId _     =>
          let localDecl ← getLocalDecl fvarId
          if localDecl.isLet && localDecl.index > (← read) then
            modify fun s => s.insert localDecl.fvarId
        | _ => pure ()
    visit (← instantiateMVars e) |>.run

  /-
    Auxiliary definition for traversing all declarations between `xs[0]` ... `xs.back` backwards.
    The `Nat` argument is the current position in the local context being visited, and it is less than
    or equal to the position of `xs.back` in the local context.
```

```
    The `Nat` context `(← read)` is the position of `xs[0]` in the local context.
  -/
  collectLetDepsAux : Nat → ReaderT Nat (StateRefT NameHashSet MetaM) Unit
    | 0   => return ()
    | i+1 => do
      if i+1 == (← read) then
        return ()
      else
        match (← getLCtx).getAt! (i+1) with
        | none => collectLetDepsAux i
        | some localDecl =>
          if (← get).contains localDecl.fvarId then
            collectLetDeclsFrom localDecl.type
            match localDecl.value? with
            | some val => collectLetDeclsFrom val
            | _ =>  pure ()
          collectLetDepsAux i

  /- Computes the set `ys`. It is a set of `FVarId`s, -/
  collectLetDeps : MetaM NameHashSet := do
    let lctx ← getLCtx
    let start := lctx.getFVar! xs[0] |>.index
    let stop  := lctx.getFVar! xs.back |>.index
    let s := xs.foldl (init := {}) fun s x => s.insert x.fvarId!
    let (_, s) ← collectLetDepsAux stop |>.run start |>.run s
    return s

  /- Computes the array `ys` containing let-decls between `xs[0]` and `xs.back` that
     some `x` in `xs` depends on. -/
  addLetDeps : MetaM (Array Expr) := do
    let lctx ← getLCtx
    let s ← collectLetDeps
    /- Convert `s` into the the array `ys` -/
    let start := lctx.getFVar! xs[0] |>.index
    let stop  := lctx.getFVar! xs.back |>.index
    let mut ys := #[]
    for i in [start:stop+1] do
      match lctx.getAt! i with
      | none => pure ()
      | some localDecl =>
        if s.contains localDecl.fvarId then
          ys := ys.push localDecl.toExpr
    return ys

/-
  Each metavariable is declared in a particular local context.
  We use the notation `C |- ?m : t` to denote a metavariable `?m` that
```

```
was declared at the local context `C` with type `t` (see `MetavarDecl`).
We also use `?m@C` as a shorthand for `C |- ?m : t` where `t` is the type of `?m`.

The following method process the unification constraint

     ?m@C a₁ ... aₙ =?= t

We say the unification constraint is a pattern IFF

  1) `a₁ ... aₙ` are pairwise distinct free variables that are <200b>*not*<200b> let-variables.
  2) `a₁ ... aₙ` are not in `C`
  3) `t` only contains free variables in `C` and/or `{a₁, ..., aₙ}`
  4) For every metavariable `?m'@C'` occurring in `t`, `C'` is a subprefix of `C`
  5) `?m` does not occur in `t`

Claim: we don't have to check free variable declarations. That is,
if `t` contains a reference to `x : A := v`, we don't need to check `v`.
Reason: The reference to `x` is a free variable, and it must be in `C` (by 1 and 3).
If `x` is in `C`, then any metavariable occurring in `v` must have been defined in a strict subprefix of `C`.
So, condition 4 and 5 are satisfied.

If the conditions above have been satisfied, then the
solution for the unification constrain is

  ?m := fun a₁ ... aₙ => t

Now, we consider some workarounds/approximations.

A1) Suppose `t` contains a reference to `x : A := v` and `x` is not in `C` (failed condition 3)
    (precise) solution: unfold `x` in `t`.

A2) Suppose some `aᵢ` is in `C` (failed condition 2)
    (approximated) solution (when `config.ctxApprox` is set to true) :
    ignore condition and also use

      ?m := fun a₁ ... aₙ => t

 Here is an example where this approximation fails:
 Given `C` containing `a : nat`, consider the following two constraints
       ?m@C a =?= a
       ?m@C b =?= a

 If we use the approximation in the first constraint, we get
       ?m := fun x => x
 when we apply this solution to the second one we get a failure.

 IMPORTANT: When applying this approximation we need to make sure the
```

```
  abstracted term `fun a₁ ... aₙ => t` is type correct. The check
  can only be skipped in the pattern case described above. Consider
  the following example. Given the local context

     (α : Type) (a : α)

  we try to solve

    ?m α =?= @id α a

  If we use the approximation above we obtain:

    ?m := (fun α' => @id α' a)

  which is a type incorrect term. `a` has type `α` but it is expected to have
  type `α'`.

  The problem occurs because the right hand side contains a free variable
  `a` that depends on the free variable `α` being abstracted. Note that
  this dependency cannot occur in patterns.

  We can address this by type checking
  the term after abstraction. This is not a significant performance
  bottleneck because this case doesn't happen very often in practice
  (262 times when compiling stdlib on Jan 2018). The second example
  is trickier, but it also occurs less frequently (8 times when compiling
  stdlib on Jan 2018, and all occurrences were at Init/Control when
  we define monads and auxiliary combinators for them).
  We considered three options for the addressing the issue on the second example:

A3) `a₁ ... aₙ` are not pairwise distinct (failed condition 1).
  In Lean3, we would try to approximate this case using an approach similar to A2.
  However, this approximation complicates the code, and is never used in the
  Lean3 stdlib and mathlib.

A4) `t` contains a metavariable `?m'@C'` where `C'` is not a subprefix of `C`.
  If `?m'` is assigned, we substitute.
  If not, we create an auxiliary metavariable with a smaller scope.
  Actually, we let `elimMVarDeps` at `MetavarContext.lean` to perform this step.

A5) If some `aᵢ` is not a free variable,
    then we use first-order unification (if `config.foApprox` is set to true)

      ?m a_1 ... a_i a_{i+1} ... a_{i+k} =?= f b_1 ... b_k

  reduces to

```

~~~
      ?M a_1 ... a_i =?= f
      a_{i+1}        =?= b_1
      ...
      a_{i+k}        =?= b_k


A6) If (m =?= v) is of the form

       ?m a_1 ... a_n =?= ?m b_1 ... b_k

    then we use first-order unification (if `config.foApprox` is set to true)

A7) When `foApprox`, we may use another approximation (`constApprox`) for solving constraints of the form
    ```
    ?m s₁ ... sₙ =?= t
    ```
    where `s₁ ... sₙ` are arbitrary terms. We solve them by assigning the constant function to `?m`.
    ```
    ?m := fun _ ... _ => t
    ```

    In general, this approximation may produce bad solutions, and may prevent coercions from being tried.
    For example, consider the term `pure (x > 0)` with inferred type `?m Prop` and expected type `IO Bool`.
    In this situation, the
    elaborator generates the unification constraint
    ```
    ?m Prop =?= IO Bool
    ```
    It is not a higher-order pattern, nor first-order approximation is applicable. However, constant approximation
    produces the bogus solution `?m := fun _ => IO Bool`, and prevents the system from using the coercion from
    the decidable proposition `x > 0` to `Bool`.

    On the other hand, the constant approximation is desirable for elaborating the term
    ```
    let f (x : _) := pure "hello"; f ()
    ```
    with expected type `IO String`.
    In this example, the following unification contraint is generated.
    ```
    ?m () String =?= IO String
    ```
    It is not a higher-order pattern, first-order approximation reduces it to
    ```
    ?m () =?= IO
    ```
    which fails to be solved. However, constant approximation solves it by assigning
    ```
~~~

```
     ?m := fun _ => IO
     ```
     Note that `f`s type is `(x : ?α) -> ?m x String`. The metavariable `?m` may depend on `x`.
     If `constApprox` is set to true, we use constant approximation. Otherwise, we use a heuristic to decide
     whether we should apply it or not. The heuristic is based on observing where the constraints above come from.
     In the first example, the constraint `?m Prop =?= IO Bool` come from polymorphic method where `?m` is expected to
     be a **function** of type `Type -> Type`. In the second example, the first argument of `?m` is used to model
     a **potential** dependency on `x`. By using constant approximation here, we are just saying the type of `f`
     does **not** depend on `x`. We claim this is a reasonable approximation in practice. Moreover, it is expected
     by any functional programmer used to non-dependently type languages (e.g., Haskell).
     We distinguish the two cases above by using the field `numScopeArgs` at `MetavarDecl`. This fiels tracks
     how many metavariable arguments are representing dependencies.
-/

def mkAuxMVar (lctx : LocalContext) (localInsts : LocalInstances) (type : Expr) (numScopeArgs : Nat := 0) : MetaM Expr := do
  mkFreshExprMVarAt lctx localInsts type MetavarKind.natural Name.anonymous numScopeArgs

namespace CheckAssignment

builtin_initialize checkAssignmentExceptionId : InternalExceptionId ← registerInternalExceptionId `checkAssignment
builtin_initialize outOfScopeExceptionId : InternalExceptionId ← registerInternalExceptionId `outOfScope

structure State where
  cache : ExprStructMap Expr := {}

structure Context where
  mvarId   : MVarId
  mvarDecl   : MetavarDecl
  fvars   : Array Expr
  hasCtxLocals : Bool
  rhs   : Expr

abbrev CheckAssignmentM := ReaderT Context $ StateRefT State MetaM

def throwCheckAssignmentFailure {α} : CheckAssignmentM α :=
  throw <| Exception.internal checkAssignmentExceptionId

def throwOutOfScopeFVar {α} : CheckAssignmentM α :=
  throw <| Exception.internal outOfScopeExceptionId

private def findCached? (e : Expr) : CheckAssignmentM (Option Expr) := do
  return (← get).cache.find? e

private def cache (e r : Expr) : CheckAssignmentM Unit := do
  modify fun s => { s with cache := s.cache.insert e r }

instance : MonadCache Expr Expr CheckAssignmentM where
```
```

```
  findCached? := findCached?
  cache       := cache

@[inline] private def visit (f : Expr → CheckAssignmentM Expr) (e : Expr) : CheckAssignmentM Expr :=
  if !e.hasExprMVar && !e.hasFVar then pure e else checkCache e (fun _ => f e)

private def addAssignmentInfo (msg : MessageData) : CheckAssignmentM MessageData := do
  let ctx ← read
  return m!"{msg} @ {mkMVar ctx.mvarId} {ctx.fvars} := {ctx.rhs}"

@[inline] def run (x : CheckAssignmentM Expr) (mvarId : MVarId) (fvars : Array Expr) (hasCtxLocals : Bool) (v : Expr) : MetaM (Option Expr) := do
  let mvarDecl ← getMVarDecl mvarId
  let ctx := { mvarId := mvarId, mvarDecl := mvarDecl, fvars := fvars, hasCtxLocals := hasCtxLocals, rhs := v : Context }
  let x : CheckAssignmentM (Option Expr) :=
    catchInternalIds [outOfScopeExceptionId, checkAssignmentExceptionId]
      (do let e ← x; return some e)
      (fun _ => pure none)
  x.run ctx |>.run' {}

mutual

  partial def checkFVar (fvar : Expr) : CheckAssignmentM Expr := do
    let ctxMeta ← readThe Meta.Context
    let ctx ← read
    if ctx.mvarDecl.lctx.containsFVar fvar then
      pure fvar
    else
      let lctx := ctxMeta.lctx
      match lctx.findFVar? fvar with
      | some (LocalDecl.ldecl (value := v) ..) => visit check v
      | _ =>
        if ctx.fvars.contains fvar then pure fvar
        else
          traceM `Meta.isDefEq.assign.outOfScopeFVar do addAssignmentInfo fvar
          throwOutOfScopeFVar

  partial def checkMVar (mvar : Expr) : CheckAssignmentM Expr := do
    let mvarId := mvar.mvarId!
    let ctx  ← read
    let mctx ← getMCtx
    if mvarId == ctx.mvarId then
      traceM `Meta.isDefEq.assign.occursCheck <| addAssignmentInfo "occurs check failed"
      throwCheckAssignmentFailure
    else match mctx.getExprAssignment? mvarId with
      | some v => check v
      | none   =>
```

```
match mctx.findDecl? mvarId with
| none          => throwUnknownMVar mvarId
| some mvarDecl =>
  if ctx.hasCtxLocals then
    throwCheckAssignmentFailure -- It is not a pattern, then we fail and fall back to FO unification
  else if mvarDecl.lctx.isSubPrefixOf ctx.mvarDecl.lctx ctx.fvars then
    /- The local context of `mvar` - free variables being abstracted is a subprefix of the metavariable being assigned.
       We "substract" variables being abstracted because we use `elimMVarDeps` -/
    pure mvar
  else if mvarDecl.depth != mctx.depth || mvarDecl.kind.isSyntheticOpaque then
    traceM `Meta.isDefEq.assign.readOnlyMVarWithBiggerLCtx <| addAssignmentInfo (mkMVar mvarId)
    throwCheckAssignmentFailure
  else
    let ctxMeta ← readThe Meta.Context
    if ctxMeta.config.ctxApprox && ctx.mvarDecl.lctx.isSubPrefixOf mvarDecl.lctx then
      /- Create an auxiliary metavariable with a smaller context and "checked" type.
         Note that `mvarType` may be different from `mvarDecl.type`. Example: `mvarType` contains
         a metavariable that we also need to reduce the context.

         We remove from `ctx.mvarDecl.lctx` any variable that is not in `mvarDecl.lctx`
         or in `ctx.fvars`. We don't need to remove the ones in `ctx.fvars` because
         `elimMVarDeps` will take care of them.

         First, we collect `toErase` the variables that need to be erased.
         Notat that if a variable is `ctx.fvars`, but it depends on variable at `toErase`,
         we must also erase it.
      -/
      let toErase := mvarDecl.lctx.foldl (init := #[]) fun toErase localDecl =>
        if ctx.mvarDecl.lctx.contains localDecl.fvarId then
          toErase
        else if ctx.fvars.any fun fvar => fvar.fvarId! == localDecl.fvarId then
          if mctx.findLocalDeclDependsOn localDecl fun fvarId => toErase.contains fvarId then
            -- localDecl depends on a variable that will be erased. So, we must add it to `toErase` too
            toErase.push localDecl.fvarId
          else
            toErase
        else
          toErase.push localDecl.fvarId
      let lctx := toErase.foldl (init := mvarDecl.lctx) fun lctx toEraseFVar =>
        lctx.erase toEraseFVar
      /- Compute new set of local instances. -/
      let localInsts := mvarDecl.localInstances.filter fun localInst => toErase.contains localInst.fvar.fvarId!
      let mvarType ← check mvarDecl.type
      let newMVar ← mkAuxMVar lctx localInsts mvarType mvarDecl.numScopeArgs
      modifyThe Meta.State fun s => { s with mctx := s.mctx.assignExpr mvarId newMVar }
      pure newMVar
    else
```

```
            traceM `Meta.isDefEq.assign.readOnlyMVarWithBiggerLCtx <| addAssignmentInfo (mkMVar mvarId)
            throwCheckAssignmentFailure

/-
  Auxiliary function used to "fix" subterms of the form `?m x_1 ... x_n` where `x_i`s are free variables,
  and one of them is out-of-scope.
  See `Expr.app` case at `check`.
  If `ctxApprox` is true, then we solve this case by creating a fresh metavariable ?n with the correct scope,
  an assigning `?m := fun _ ... _ => ?n` -/
partial def assignToConstFun (mvar : Expr) (numArgs : Nat) (newMVar : Expr) : MetaM Bool := do
  let mvarType ← inferType mvar
  forallBoundedTelescope mvarType numArgs fun xs _ => do
    if xs.size != numArgs then pure false
    else
      let some v ← mkLambdaFVarsWithLetDeps xs newMVar | return false
      match (← checkAssignmentAux mvar.mvarId! #[] false v) with
      | some v => checkTypesAndAssign mvar v
      | none   => return false

-- See checkAssignment
partial def checkAssignmentAux (mvarId : MVarId) (fvars : Array Expr) (hasCtxLocals : Bool) (v : Expr) : MetaM (Option Expr) := do
  run (check v) mvarId fvars hasCtxLocals v

partial def check (e : Expr) : CheckAssignmentM Expr := do
  match e with
  | Expr.mdata _ b _     => return e.updateMData! (← visit check b)
  | Expr.proj _ _ s _    => return e.updateProj! (← visit check s)
  | Expr.lam _ d b _     => return e.updateLambdaE! (← visit check d) (← visit check b)
  | Expr.forallE _ d b _ => return e.updateForallE! (← visit check d) (← visit check b)
  | Expr.letE _ t v b _  => return e.updateLet! (← visit check t) (← visit check v) (← visit check b)
  | Expr.bvar ..         => return e
  | Expr.sort ..         => return e
  | Expr.const ..        => return e
  | Expr.lit ..          => return e
  | Expr.fvar ..         => visit checkFVar e
  | Expr.mvar ..         => visit checkMVar e
  | Expr.app ..          => e.withApp fun f args => do
    let ctxMeta ← readThe Meta.Context
    if f.isMVar && ctxMeta.config.ctxApprox && args.all Expr.isFVar then
      let f ← visit checkMVar f
      catchInternalId outOfScopeExceptionId
        (do
          let args ← args.mapM (visit check)
          return mkAppN f args)
        (fun ex => do
          if !f.isMVar then
            throw ex
```

```
          else if (← isDelayedAssigned f.mvarId!) then
            throw ex
          else
            let eType ← inferType e
            let mvarType ← check eType
            /- Create an auxiliary metavariable with a smaller context and "checked" type, assign `?f := fun _ => ?newMVar`
                 Note that `mvarType` may be different from `eType`. -/
            let ctx ← read
            let newMVar ← mkAuxMVar ctx.mvarDecl.lctx ctx.mvarDecl.localInstances mvarType
            if (← assignToConstFun f args.size newMVar) then
              pure newMVar
            else
              throw ex)
    else
      let f ← visit check f
      let args ← args.mapM (visit check)
      return mkAppN f args

end

end CheckAssignment

namespace CheckAssignmentQuick

partial def check
    (hasCtxLocals ctxApprox : Bool)
    (mctx : MetavarContext) (lctx : LocalContext) (mvarDecl : MetavarDecl) (mvarId : MVarId) (fvars : Array Expr) (e : Expr) : Bool :=
  let rec visit (e : Expr) : Bool :=
    if !e.hasExprMVar && !e.hasFVar then
      true
    else match e with
    | Expr.mdata _ b _     => visit b
    | Expr.proj _ _ s _    => visit s
    | Expr.app f a _       => visit f && visit a
    | Expr.lam _ d b _     => visit d && visit b
    | Expr.forallE _ d b _ => visit d && visit b
    | Expr.letE _ t v b _  => visit t && visit v && visit b
    | Expr.bvar ..         => true
    | Expr.sort ..         => true
    | Expr.const ..        => true
    | Expr.lit ..          => true
    | Expr.fvar fvarId ..  =>
      if mvarDecl.lctx.contains fvarId then true
      else match lctx.find? fvarId with
        | some (LocalDecl.ldecl (value := v) ..) => false -- need expensive CheckAssignment.check
        | _ =>
          if fvars.any fun x => x.fvarId! == fvarId then true
```

```
          else false -- We could throw an exception here, but we would have to use ExceptM. So, we let CheckAssignment.check do it
    | Expr.mvar mvarId' _  =>
      match mctx.getExprAssignment? mvarId' with
      | some _ => false -- use CheckAssignment.check to instantiate
      | none   =>
        if mvarId' == mvarId then false -- occurs check failed, use CheckAssignment.check to throw exception
        else match mctx.findDecl? mvarId' with
          | none           => false
          | some mvarDecl' =>
            if hasCtxLocals then false -- use CheckAssignment.check
            else if mvarDecl'.lctx.isSubPrefixOf mvarDecl.lctx fvars then true
            else false -- use CheckAssignment.check
  visit e

end CheckAssignmentQuick

/--
  Auxiliary function for handling constraints of the form `?m a₁ ... aₙ =?= v`.
  It will check whether we can perform the assignment
  ```
  ?m := fun fvars => v
  ```
  The result is `none` if the assignment can't be performed.
  The result is `some newV` where `newV` is a possibly updated `v`. This method may need
  to unfold let-declarations. -/
def checkAssignment (mvarId : MVarId) (fvars : Array Expr) (v : Expr) : MetaM (Option Expr) := do
  /- Check whether `mvarId` occurs in the type of `fvars` or not. If it does, return `none`
     to prevent us from creating the cyclic assignment `?m := fun fvars => v` -/
  for fvar in fvars do
    unless (← occursCheck mvarId (← inferType fvar)) do
      return none
  if !v.hasExprMVar && !v.hasFVar then
    pure (some v)
  else
    let mvarDecl ← getMVarDecl mvarId
    let hasCtxLocals := fvars.any fun fvar => mvarDecl.lctx.containsFVar fvar
    let ctx ← read
    let mctx ← getMCtx
    if CheckAssignmentQuick.check hasCtxLocals ctx.config.ctxApprox mctx ctx.lctx mvarDecl mvarId fvars v then
      pure (some v)
    else
      let v ← instantiateMVars v
      CheckAssignment.checkAssignmentAux mvarId fvars hasCtxLocals v

private def processAssignmentFOApproxAux (mvar : Expr) (args : Array Expr) (v : Expr) : MetaM Bool :=
  match v with
  | Expr.app f a _ =>
```

```
    if args.isEmpty then
      pure false
    else
      Meta.isExprDefEqAux args.back a <&&> Meta.isExprDefEqAux (mkAppRange mvar 0 (args.size - 1) args) f
  | _              => pure false

/-
  Auxiliary method for applying first-order unification. It is an approximation.
  Remark: this method is trying to solve the unification constraint:

      ?m a₁ ... aₙ =?= v

   It is uses processAssignmentFOApproxAux, if it fails, it tries to unfold `v`.

   We have added support for unfolding here because we want to be able to solve unification problems such as

      ?m Unit =?= ITactic

   where `ITactic` is defined as

   def ITactic := Tactic Unit
-/
private partial def processAssignmentFOApprox (mvar : Expr) (args : Array Expr) (v : Expr) : MetaM Bool :=
  let rec loop (v : Expr) := do
    let cfg ← getConfig
    if !cfg.foApprox then
      pure false
    else
      trace[Meta.isDefEq.foApprox]! "{mvar} {args} := {v}"
      let v := v.headBeta
      if (← commitWhen <| processAssignmentFOApproxAux mvar args v) then
        pure true
      else
        match (← unfoldDefinition? v) with
        | none   => pure false
        | some v => loop v
  loop v

private partial def simpAssignmentArgAux : Expr → MetaM Expr
  | Expr.mdata _ e _       => simpAssignmentArgAux e
  | e@(Expr.fvar fvarId _) => do
    let decl ← getLocalDecl fvarId
    match decl.value? with
    | some value => simpAssignmentArgAux value
    | _          => pure e
  | e => pure e
```

```
/- Auxiliary procedure for processing `?m a₁ ... aₙ =?= v`.
   We apply it to each `aᵢ`. It instantiates assigned metavariables if `aᵢ` is of the form `f[?n] b₁ ... bₘ`,
   and then removes metadata, and zeta-expand let-decls. -/
private def simpAssignmentArg (arg : Expr) : MetaM Expr := do
  let arg ← if arg.getAppFn.hasExprMVar then instantiateMVars arg else pure arg
  simpAssignmentArgAux arg

/- Assign `mvar := fun a_1 ... a_{numArgs} => v`.
   We use it at `processConstApprox` and `isDefEqMVarSelf` -/
private def assignConst (mvar : Expr) (numArgs : Nat) (v : Expr) : MetaM Bool := do
  let mvarDecl ← getMVarDecl mvar.mvarId!
  forallBoundedTelescope mvarDecl.type numArgs fun xs _ => do
    if xs.size != numArgs then
      pure false
    else
      let some v ← mkLambdaFVarsWithLetDeps xs v | pure false
      match (← checkAssignment mvar.mvarId! #[] v) with
      | none   => pure false
      | some v =>
        trace[Meta.isDefEq.constApprox]! "{mvar} := {v}"
        checkTypesAndAssign mvar v

private def processConstApprox (mvar : Expr) (numArgs : Nat) (v : Expr) : MetaM Bool := do
  let cfg ← getConfig
  let mvarId := mvar.mvarId!
  let mvarDecl ← getMVarDecl mvarId
  if mvarDecl.numScopeArgs == numArgs || cfg.constApprox then
    assignConst mvar numArgs v
  else
    pure false

/-- Tries to solve `?m a₁ ... aₙ =?= v` by assigning `?m`.
    It assumes `?m` is unassigned. -/
private partial def processAssignment (mvarApp : Expr) (v : Expr) : MetaM Bool :=
  traceCtx `Meta.isDefEq.assign do
    trace[Meta.isDefEq.assign]! "{mvarApp} := {v}"
    let mvar := mvarApp.getAppFn
    let mvarDecl ← getMVarDecl mvar.mvarId!
    let rec process (i : Nat) (args : Array Expr) (v : Expr) := do
      let cfg ← getConfig
      let useFOApprox (args : Array Expr) : MetaM Bool :=
        processAssignmentFOApprox mvar args v <||> processConstApprox mvar args.size v
      if h : i < args.size then
        let arg := args.get ⟨i, h⟩
        let arg ← simpAssignmentArg arg
        let args := args.set ⟨i, h⟩ arg
        match arg with
```

```lean
      | Expr.fvar fvarId _ =>
        if args[0:i].any fun prevArg => prevArg == arg then
          useFOApprox args
        else if mvarDecl.lctx.contains fvarId && !cfg.quasiPatternApprox then
          useFOApprox args
        else
          process (i+1) args v
      | _ =>
        useFOApprox args
    else
      let v ← instantiateMVars v -- enforce A4
      if v.getAppFn == mvar then
        -- using A6
        useFOApprox args
      else
        let mvarId := mvar.mvarId!
        match (← checkAssignment mvarId args v) with
        | none   => useFOApprox args
        | some v => do
          trace[Meta.isDefEq.assign.beforeMkLambda]! "{mvar} {args} := {v}"
          let some v ← mkLambdaFVarsWithLetDeps args v | return false
          if args.any (fun arg => mvarDecl.lctx.containsFVar arg) then
            /- We need to type check `v` because abstraction using `mkLambdaFVars` may have produced
               a type incorrect term. See discussion at A2 -/
            if (← isTypeCorrect v) then
              checkTypesAndAssign mvar v
            else
              trace[Meta.isDefEq.assign.typeError]! "{mvar} := {v}"
              useFOApprox args
          else
            checkTypesAndAssign mvar v
  process 0 mvarApp.getAppArgs v

/--
  Similar to processAssignment, but if it fails, compute v's whnf and try again.
  This helps to solve constraints such as `?m =?= { α := ?m, ... }.α`
  Note this is not perfect solution since we still fail occurs check for constraints such as
  ```lean
    ?m =?= List { α := ?m, β := Nat }.β
  ```
-/
private def processAssignment' (mvarApp : Expr) (v : Expr) : MetaM Bool := do
  if (← processAssignment mvarApp v) then
    return true
  else
    let vNew ← whnf v
    if vNew != v then
```

```
      if mvarApp == vNew then
        return true
      else
        processAssignment mvarApp vNew
    else
      return false

private def isDeltaCandidate? (t : Expr) : MetaM (Option ConstantInfo) := do
  match t.getAppFn with
  | Expr.const c _ _ =>
    match (← getConst? c) with
    | r@(some info) => if info.hasValue then return r else return none
    | _             => return none
  | _ => pure none

/-- Auxiliary method for isDefEqDelta -/
private def isListLevelDefEq (us vs : List Level) : MetaM LBool :=
  toLBoolM <| isListLevelDefEqAux us vs

/-- Auxiliary method for isDefEqDelta -/
private def isDefEqLeft (fn : Name) (t s : Expr) : MetaM LBool := do
  trace[Meta.isDefEq.delta.unfoldLeft]! fn
  toLBoolM <| Meta.isExprDefEqAux t s

/-- Auxiliary method for isDefEqDelta -/
private def isDefEqRight (fn : Name) (t s : Expr) : MetaM LBool := do
  trace[Meta.isDefEq.delta.unfoldRight]! fn
  toLBoolM <| Meta.isExprDefEqAux t s

/-- Auxiliary method for isDefEqDelta -/
private def isDefEqLeftRight (fn : Name) (t s : Expr) : MetaM LBool := do
  trace[Meta.isDefEq.delta.unfoldLeftRight]! fn
  toLBoolM <| Meta.isExprDefEqAux t s

/-- Try to solve `f a₁ ... aₙ =?= f b₁ ... bₙ` by solving `a₁ =?= b₁, ..., aₙ =?= bₙ`.

    Auxiliary method for isDefEqDelta -/
private def tryHeuristic (t s : Expr) : MetaM Bool :=
  let tFn := t.getAppFn
  let sFn := s.getAppFn
  traceCtx `Meta.isDefEq.delta do
    /-
      We process arguments before universe levels to reduce a source of brittleness in the TC procedure.

      In the TC procedure, we can solve problems containing metavariables.
      If the TC procedure tries to assign one of these metavariables, it interrupts the search
      using a "stuck" exception. The elaborator catches it, and "interprets" it as "we should try again later".
```

```
    Now suppose we have a TC problem, and there are two "local" candidate instances we can try: "bad" and "good".
    The "bad" candidate is stuck because of a universe metavariable in the TC problem.
    If we try "bad" first, the TC procedure is interrupted. Moreover, if we have ignored the exception,
    "bad" would fail anyway trying to assign two different free variables `α =?= β`.
    Example: `Preorder.{?u} α =?= Preorder.{?v} β`, where `?u` and `?v` are universe metavariables that were
    not created by the TC procedure.
    The key issue here is that we have an `isDefEq t s` invocation that is interrupted by the "stuck" exception,
    but it would have failed anyway if we had continued processing it.
    By solving the arguments first, we make the example above fail without throwing the "stuck" exception.

    TODO: instead of throwing an exception as soon as we get stuck, we should just set a flag.
    Then the entry-point for `isDefEq` checks the flag before returning `true`.
  -/
  commitWhen do
    let b ← isDefEqArgs tFn t.getAppArgs s.getAppArgs
            <&&>
            isListLevelDefEqAux tFn.constLevels! sFn.constLevels!
    unless b do
      trace[Meta.isDefEq.delta]! "heuristic failed {t} =?= {s}"
    pure b

/-- Auxiliary method for isDefEqDelta -/
private abbrev unfold {α} (e : Expr) (failK : MetaM α) (successK : Expr → MetaM α) : MetaM α := do
  match (← unfoldDefinition? e) with
  | some e => successK e
  | none   => failK

/-- Auxiliary method for isDefEqDelta -/
private def unfoldBothDefEq (fn : Name) (t s : Expr) : MetaM LBool := do
  match t, s with
  | Expr.const _ ls₁ _, Expr.const _ ls₂ _ => isListLevelDefEq ls₁ ls₂
  | Expr.app _ _ _,     Expr.app _ _ _     =>
    if (← tryHeuristic t s) then
      pure LBool.true
    else
      unfold t
       (unfold s (pure LBool.false) (fun s => isDefEqRight fn t s))
       (fun t => unfold s (isDefEqLeft fn t s) (fun s => isDefEqLeftRight fn t s))
  | _, _ => pure LBool.false

private def sameHeadSymbol (t s : Expr) : Bool :=
  match t.getAppFn, s.getAppFn with
  | Expr.const c₁ _ _, Expr.const c₂ _ _ => true
  | _,                 _                 => false

/--
  - If headSymbol (unfold t) == headSymbol s, then unfold t
```

```
  - If headSymbol (unfold s) == headSymbol t, then unfold s
  - Otherwise unfold t and s if possible.

  Auxiliary method for isDefEqDelta -/
private def unfoldComparingHeadsDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool :=
  unfold t
    (unfold s
      (pure LBool.undef) -- `t` and `s` failed to be unfolded
      (fun s => isDefEqRight sInfo.name t s))
    (fun tNew =>
      if sameHeadSymbol tNew s then
        isDefEqLeft tInfo.name tNew s
      else
        unfold s
          (isDefEqLeft tInfo.name tNew s)
          (fun sNew =>
            if sameHeadSymbol t sNew then
              isDefEqRight sInfo.name t sNew
            else
              isDefEqLeftRight tInfo.name tNew sNew))

/-- If `t` and `s` do not contain metavariables, then use
    kernel definitional equality heuristics.
    Otherwise, use `unfoldComparingHeadsDefEq`.

    Auxiliary method for isDefEqDelta -/
private def unfoldDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool :=
  if !t.hasExprMVar && !s.hasExprMVar then
    /- If `t` and `s` do not contain metavariables,
       we simulate strategy used in the kernel. -/
    if tInfo.hints.lt sInfo.hints then
      unfold t (unfoldComparingHeadsDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
    else if sInfo.hints.lt tInfo.hints then
      unfold s (unfoldComparingHeadsDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
    else
      unfoldComparingHeadsDefEq tInfo sInfo t s
  else
    unfoldComparingHeadsDefEq tInfo sInfo t s

/--
  When `TransparencyMode` is set to `default` or `all`.
  If `t` is reducible and `s` is not ==> `isDefEqLeft  (unfold t) s`
  If `s` is reducible and `t` is not ==> `isDefEqRight t (unfold s)`

  Otherwise, use `unfoldDefEq`

  Auxiliary method for isDefEqDelta -/
```

```
private def unfoldReducibeDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool := do
  if (← shouldReduceReducibleOnly) then
    unfoldDefEq tInfo sInfo t s
  else
    let tReducible ← isReducible tInfo.name
    let sReducible ← isReducible sInfo.name
    if tReducible && !sReducible then
      unfold t (unfoldDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
    else if !tReducible && sReducible then
      unfold s (unfoldDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
    else
      unfoldDefEq tInfo sInfo t s

/--
  If `t` is a projection function application and `s` is not ==> `isDefEqRight t (unfold s)`
  If `s` is a projection function application and `t` is not ==> `isDefEqRight (unfold t) s`

  Otherwise, use `unfoldReducibeDefEq`

  Auxiliary method for isDefEqDelta -/
private def unfoldNonProjFnDefEq (tInfo sInfo : ConstantInfo) (t s : Expr) : MetaM LBool := do
  let tProj? ← isProjectionFn tInfo.name
  let sProj? ← isProjectionFn sInfo.name
  if tProj? && !sProj? then
    unfold s (unfoldDefEq tInfo sInfo t s) fun s => isDefEqRight sInfo.name t s
  else if !tProj? && sProj? then
    unfold t (unfoldDefEq tInfo sInfo t s) fun t => isDefEqLeft tInfo.name t s
  else
    unfoldReducibeDefEq tInfo sInfo t s

/--
  isDefEq by lazy delta reduction.
  This method implements many different heuristics:
  1- If only `t` can be unfolded => then unfold `t` and continue
  2- If only `s` can be unfolded => then unfold `s` and continue
  3- If `t` and `s` can be unfolded and they have the same head symbol, then
     a) First try to solve unification by unifying arguments.
     b) If it fails, unfold both and continue.
     Implemented by `unfoldBothDefEq`
  4- If `t` is a projection function application and `s` is not => then unfold `s` and continue.
  5- If `s` is a projection function application and `t` is not => then unfold `t` and continue.
  Remark: 4&5 are implemented by `unfoldNonProjFnDefEq`
  6- If `t` is reducible and `s` is not => then unfold `t` and continue.
  7- If `s` is reducible and `t` is not => then unfold `s` and continue
  Remark: 6&7 are implemented by `unfoldReducibeDefEq`
  8- If `t` and `s` do not contain metavariables, then use heuristic used in the Kernel.
     Implemented by `unfoldDefEq`
```

```
  9- If `headSymbol (unfold t) == headSymbol s`, then unfold t and continue.
  10- If `headSymbol (unfold s) == headSymbol t`, then unfold s
  11- Otherwise, unfold `t` and `s` and continue.
  Remark: 9&10&11 are implemented by `unfoldComparingHeadsDefEq` -/
private def isDefEqDelta (t s : Expr) : MetaM LBool := do
  let tInfo? ← isDeltaCandidate? t.getAppFn
  let sInfo? ← isDeltaCandidate? s.getAppFn
  match tInfo?, sInfo? with
  | none,       none       => pure LBool.undef
  | some tInfo, none       => unfold t (pure LBool.undef) fun t => isDefEqLeft tInfo.name t s
  | none,       some sInfo => unfold s (pure LBool.undef) fun s => isDefEqRight sInfo.name t s
  | some tInfo, some sInfo =>
    if tInfo.name == sInfo.name then
      unfoldBothDefEq tInfo.name t s
    else
      unfoldNonProjFnDefEq tInfo sInfo t s

private def isAssigned : Expr → MetaM Bool
  | Expr.mvar mvarId _ => isExprMVarAssigned mvarId
  | _                  => pure false

private def isDelayedAssignedHead (tFn : Expr) (t : Expr) : MetaM Bool := do
  match tFn with
  | Expr.mvar mvarId _ =>
    if (← isDelayedAssigned mvarId) then
      let tNew ← instantiateMVars t
      return tNew != t
    else
      pure false
  | _ => pure false

private def isSynthetic : Expr → MetaM Bool
  | Expr.mvar mvarId _ => do
    let mvarDecl ← getMVarDecl mvarId
    match mvarDecl.kind with
    | MetavarKind.synthetic       => pure true
    | MetavarKind.syntheticOpaque => pure true
    | MetavarKind.natural         => pure false
  | _                  => pure false

private def isAssignable : Expr → MetaM Bool
  | Expr.mvar mvarId _ => do let b ← isReadOnlyOrSyntheticOpaqueExprMVar mvarId; pure (!b)
  | _                  => pure false

private def etaEq (t s : Expr) : Bool :=
  match t.etaExpanded? with
  | some t => t == s
```

```
  | none   => false

private def isLetFVar (fvarId : FVarId) : MetaM Bool := do
  let decl ← getLocalDecl fvarId
  pure decl.isLet

private def isDefEqProofIrrel (t s : Expr) : MetaM LBool := do
  let status ← isProofQuick t
  match status with
  | LBool.false =>
    pure LBool.undef
  | LBool.true  =>
    let tType ← inferType t
    let sType ← inferType s
    toLBoolM <| Meta.isExprDefEqAux tType sType
  | LBool.undef =>
    let tType ← inferType t
    if (← isProp tType) then
      let sType ← inferType s
      toLBoolM <| Meta.isExprDefEqAux tType sType
    else
      pure LBool.undef

/- Try to solve constraint of the form `?m args₁ =?= ?m args₂`.
   - First try to unify `args₁` and `args₂`, and return true if successful
   - Otherwise, try to assign `?m` to a constant function of the form `fun x_1 ... x_n => ?n`
     where `?n` is a fresh metavariable. See `processConstApprox`. -/
private def isDefEqMVarSelf (mvar : Expr) (args₁ args₂ : Array Expr) : MetaM Bool := do
  if args₁.size != args₂.size then
    pure false
  else if (← isDefEqArgs mvar args₁ args₂) then
    pure true
  else if !(← isAssignable mvar) then
    pure false
  else
    let cfg ← getConfig
    let mvarId := mvar.mvarId!
    let mvarDecl ← getMVarDecl mvarId
    if mvarDecl.numScopeArgs == args₁.size || cfg.constApprox then
      let type ← inferType (mkAppN mvar args₁)
      let auxMVar ← mkAuxMVar mvarDecl.lctx mvarDecl.localInstances type
      assignConst mvar args₁.size auxMVar
    else
      pure false

/- Remove unnecessary let-decls -/
private def consumeLet : Expr → Expr
```

```
  | e@(Expr.letE _ _ _ b _) => if b.hasLooseBVars then e else consumeLet b
  | e                       => e

mutual

private partial def isDefEqQuick (t s : Expr) : MetaM LBool :=
  let t := consumeLet t
  let s := consumeLet s
  match t, s with
  | Expr.lit  l₁ _,      Expr.lit l₂ _       => return (l₁ == l₂).toLBool
  | Expr.sort u _,       Expr.sort v _       => toLBoolM <| isLevelDefEqAux u v
  | Expr.lam ..,         Expr.lam ..         => if t == s then pure LBool.true else toLBoolM <| isDefEqBinding t s
  | Expr.forallE ..,     Expr.forallE ..     => if t == s then pure LBool.true else toLBoolM <| isDefEqBinding t s
  | Expr.mdata _ t _,    s                   => isDefEqQuick t s
  | t,                   Expr.mdata _ s _    => isDefEqQuick t s
  | Expr.fvar fvarId₁ _, Expr.fvar fvarId₂ _ => do
    if (← isLetFVar fvarId₁ <||> isLetFVar fvarId₂) then
      pure LBool.undef
    else if fvarId₁ == fvarId₂ then
      pure LBool.true
    else
      isDefEqProofIrrel t s
  | t, s =>
    isDefEqQuickOther t s

private partial def isDefEqQuickOther (t s : Expr) : MetaM LBool := do
  if t == s then
    pure LBool.true
  else if etaEq t s || etaEq s t then
    pure LBool.true  -- t =?= (fun xs => t xs)
  else
    let tFn := t.getAppFn
    let sFn := s.getAppFn
    if !tFn.isMVar && !sFn.isMVar then
      pure LBool.undef
    else if (← isAssigned tFn) then
      let t ← instantiateMVars t
      isDefEqQuick t s
    else if (← isAssigned sFn) then
      let s ← instantiateMVars s
      isDefEqQuick t s
    else if (← isDelayedAssignedHead tFn t) then
      let t ← instantiateMVars t
      isDefEqQuick t s
    else if (← isDelayedAssignedHead sFn s) then
      let s ← instantiateMVars s
      isDefEqQuick t s
```

```
    else if (← isSynthetic tFn <&&> trySynthPending tFn) then
      let t ← instantiateMVars t
      isDefEqQuick t s
    else if (← isSynthetic sFn <&&> trySynthPending sFn) then
      let s ← instantiateMVars s
      isDefEqQuick t s
    else if tFn.isMVar && sFn.isMVar && tFn == sFn then
      Bool.toLBool <$> isDefEqMVarSelf tFn t.getAppArgs s.getAppArgs
    else
      let tAssign? ← isAssignable tFn
      let sAssign? ← isAssignable sFn
      let assignableMsg (b : Bool) := if b then "[assignable]" else "[nonassignable]"
      trace[Meta.isDefEq]! "{t} {assignableMsg tAssign?} =?= {s} {assignableMsg sAssign?}"
      if tAssign? && !sAssign? then
        toLBoolM <| processAssignment' t s
      else if !tAssign? && sAssign? then
        toLBoolM <| processAssignment' s t
      else if !tAssign? && !sAssign? then
        if tFn.isMVar || sFn.isMVar then
          let ctx ← read
          if ctx.config.isDefEqStuckEx then do
            trace[Meta.isDefEq.stuck]! "{t} =?= {s}"
            Meta.throwIsDefEqStuck
          else
            pure LBool.false
        else
          pure LBool.undef
      else
        isDefEqQuickMVarMVar t s

-- Both `t` and `s` are terms of the form `?m ...`
private partial def isDefEqQuickMVarMVar (t s : Expr) : MetaM LBool := do
  let tFn := t.getAppFn
  let sFn := s.getAppFn
  let tMVarDecl ← getMVarDecl tFn.mvarId!
  let sMVarDecl ← getMVarDecl sFn.mvarId!
  if s.isMVar && !t.isMVar then
     /- Solve `?m t =?= ?n` by trying first `?n := ?m t`.
        Reason: this assignment is precise. -/
     if (← commitWhen (processAssignment s t)) then
       pure LBool.true
     else
       toLBoolM <| processAssignment t s
  else
     if (← commitWhen (processAssignment t s)) then
       pure LBool.true
     else
```

```
      toLBoolM <| processAssignment s t

end

@[inline] def whenUndefDo (x : MetaM LBool) (k : MetaM Bool) : MetaM Bool := do
  let status ← x
  match status with
  | LBool.true  => pure true
  | LBool.false => pure false
  | LBool.undef => k

@[specialize] private def unstuckMVar (e : Expr) (successK : Expr → MetaM Bool) (failK : MetaM Bool): MetaM Bool := do
  match (← getStuckMVar? e) with
  | some mvarId =>
    trace[Meta.isDefEq.stuckMVar]! "found stuck MVar {mkMVar mvarId} : {← inferType (mkMVar mvarId)}"
    if (← Meta.synthPending mvarId) then
      let e ← instantiateMVars e
      successK e
    else
      failK
  | none   => failK

private def isDefEqOnFailure (t s : Expr) : MetaM Bool :=
  unstuckMVar t (fun t => Meta.isExprDefEqAux t s) <|
  unstuckMVar s (fun s => Meta.isExprDefEqAux t s) <|
  tryUnificationHints t s <||> tryUnificationHints s t

private def isDefEqProj : Expr → Expr → MetaM Bool
  | Expr.proj _ i t _, Expr.proj _ j s _ => pure (i == j) <&&> Meta.isExprDefEqAux t s
  | _, _ => pure false

/-
  Given applications `t` and `s` that are in WHNF (modulo the current transparency setting),
  check whether they are definitionally equal or not.
-/
private def isDefEqApp (t s : Expr) : MetaM Bool := do
  let tFn := t.getAppFn
  let sFn := s.getAppFn
  if tFn.isConst && sFn.isConst && tFn.constName! == sFn.constName! then
    /- See comment at `tryHeuristic` explaining why we processe arguments before universe levels. -/
    if (← commitWhen (isDefEqArgs tFn t.getAppArgs s.getAppArgs <&&> isListLevelDefEqAux tFn.constLevels! sFn.constLevels!)) then
      return true
    else
      isDefEqOnFailure t s
  else if (← commitWhen (Meta.isExprDefEqAux tFn s.getAppFn <&&> isDefEqArgs tFn t.getAppArgs s.getAppArgs)) then
    return true
  else
```

```
    isDefEqOnFailure t s

partial def isExprDefEqAuxImpl (t : Expr) (s : Expr) : MetaM Bool := do
  trace[Meta.isDefEq.step]! "{t} =?= {s}"
  checkMaxHeartbeats "isDefEq"
  withNestedTraces do
  whenUndefDo (isDefEqQuick t s) do
  whenUndefDo (isDefEqProofIrrel t s) do
  let t' ← whnfCore t
  let s' ← whnfCore s
  if t != t' || s != s' then
    isExprDefEqAuxImpl t' s'
  else do
    if (← (isDefEqEta t s <||> isDefEqEta s t)) then pure true else
    if (← isDefEqProj t s) then pure true else
    whenUndefDo (isDefEqNative t s) do
    whenUndefDo (isDefEqNat t s) do
    whenUndefDo (isDefEqOffset t s) do
    whenUndefDo (isDefEqDelta t s) do
    if t.isConst && s.isConst then
      if t.constName! == s.constName! then isListLevelDefEqAux t.constLevels! s.constLevels! else pure false
    else if t.isApp && s.isApp then
      isDefEqApp t s
    else
      whenUndefDo (isDefEqStringLit t s) do
      isDefEqOnFailure t s

builtin_initialize
  isExprDefEqAuxRef.set isExprDefEqAuxImpl

builtin_initialize
  registerTraceClass `Meta.isDefEq
  registerTraceClass `Meta.isDefEq.foApprox
  registerTraceClass `Meta.isDefEq.constApprox
  registerTraceClass `Meta.isDefEq.delta
  registerTraceClass `Meta.isDefEq.step
  registerTraceClass `Meta.isDefEq.assign

end Lean.Meta
// ::::::::::::
// ForEachExpr.lean
// ::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
```

```
import Lean.Expr
import Lean.Util.MonadCache
import Lean.Meta.Basic

namespace Lean.Meta
namespace ForEachExpr

abbrev M := MonadCacheT Expr Unit MetaM

mutual

private partial def visitBinder (fn : Expr → MetaM Bool) : Array Expr → Nat → Expr → M Unit
  | fvars, j, Expr.lam n d b c => do
    let d := d.instantiateRevRange j fvars.size fvars;
    visit fn d;
    withLocalDecl n c.binderInfo d fun x =>
      visitBinder fn (fvars.push x) j b
  | fvars, j, Expr.forallE n d b c => do
    let d := d.instantiateRevRange j fvars.size fvars;
    visit fn d;
    withLocalDecl n c.binderInfo d fun x =>
      visitBinder fn (fvars.push x) j b
  | fvars, j, Expr.letE n t v b _ => do
    let t := t.instantiateRevRange j fvars.size fvars;
    visit fn t;
    let v := v.instantiateRevRange j fvars.size fvars;
    visit fn v;
    withLetDecl n t v fun x =>
      visitBinder fn (fvars.push x) j b
  | fvars, j, e => visit fn $ e.instantiateRevRange j fvars.size fvars

partial def visit (fn : Expr → MetaM Bool) (e : Expr) : M Unit :=
  checkCache e fun _ => do
    if (← liftM (fn e)) then
      match e with
      | Expr.forallE _ _ _ _   => visitBinder fn #[] 0 e
      | Expr.lam _ _ _ _       => visitBinder fn #[] 0 e
      | Expr.letE _ _ _ _ _    => visitBinder fn #[] 0 e
      | Expr.app f a _         => visit fn f; visit fn a
      | Expr.mdata _ b _       => visit fn b
      | Expr.proj _ _ b _      => visit fn b
      | _                      => pure ()

end

end ForEachExpr
```

```
/-- Similar to `Expr.forEach'`, but creates free variables whenever going inside of a binder. -/
def forEachExpr' (e : Expr) (f : Expr → MetaM Bool) : MetaM Unit :=
  ForEachExpr.visit f e |>.run

/-- Similar to `Expr.forEach`, but creates free variables whenever going inside of a binder. -/
def forEachExpr (e : Expr) (f : Expr → MetaM Unit) : MetaM Unit :=
  forEachExpr' e fun e => do
    f e
    pure true

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

namespace Lean.Meta

@[inline] private def checkFunInfoCache (fn : Expr) (maxArgs? : Option Nat) (k : MetaM FunInfo) : MetaM FunInfo := do
  let s ← get
  let t ← getTransparency
  match s.cache.funInfo.find? ⟨t, fn, maxArgs?⟩ with
  | some finfo => pure finfo
  | none       => do
    let finfo ← k
    modify fun s => { s with cache := { s.cache with funInfo := s.cache.funInfo.insert ⟨t, fn, maxArgs?⟩ finfo } }
    pure finfo

@[inline] private def whenHasVar {α} (e : Expr) (deps : α) (k : α → α) : α :=
  if e.hasFVar then k deps else deps

private def collectDeps (fvars : Array Expr) (e : Expr) : Array Nat :=
  let rec visit : Expr → Array Nat → Array Nat
    | e@(Expr.app f a _),       deps => whenHasVar e deps (visit a ∘ visit f)
    | e@(Expr.forallE _ d b _), deps => whenHasVar e deps (visit b ∘ visit d)
    | e@(Expr.lam _ d b _),     deps => whenHasVar e deps (visit b ∘ visit d)
    | e@(Expr.letE _ t v b _),  deps => whenHasVar e deps (visit b ∘ visit v ∘ visit t)
    | Expr.proj _ _ e _,        deps => visit e deps
    | Expr.mdata _ e _,         deps => visit e deps
    | e@(Expr.fvar _ _),        deps =>
      match fvars.indexOf? e with
```

```
      | none   => deps
      | some i => if deps.contains i.val then deps else deps.push i.val
    | _,                        deps => deps
  let deps := visit e #[]
  deps.qsort (fun i j => i < j)

/-- Update `hasFwdDeps` fields using new `backDeps` -/
private def updateHasFwdDeps (pinfo : Array ParamInfo) (backDeps : Array Nat) : Array ParamInfo :=
  if backDeps.size == 0 then
    pinfo
  else
    -- update hasFwdDeps fields
    pinfo.mapIdx fun i info =>
      if info.hasFwdDeps then info
      else if backDeps.contains i then
        { info with hasFwdDeps := true }
      else
        info

private def getFunInfoAux (fn : Expr) (maxArgs? : Option Nat) : MetaM FunInfo :=
  checkFunInfoCache fn maxArgs? do
    let fnType ← inferType fn
    withTransparency TransparencyMode.default do
      forallBoundedTelescope fnType maxArgs? fun fvars type => do
        let mut pinfo := #[]
        for i in [:fvars.size] do
          let fvar := fvars[i]
          let decl ← getFVarLocalDecl fvar
          let backDeps := collectDeps fvars decl.type
          pinfo := updateHasFwdDeps pinfo backDeps
          pinfo := pinfo.push {
            backDeps     := backDeps,
            implicit     := decl.binderInfo == BinderInfo.implicit,
            instImplicit := decl.binderInfo == BinderInfo.instImplicit
          }
        let resultDeps := collectDeps fvars type
        let pinfo      := updateHasFwdDeps pinfo resultDeps
        pure { resultDeps := resultDeps, paramInfo := pinfo }

def getFunInfo (fn : Expr) : MetaM FunInfo :=
  getFunInfoAux fn none

def getFunInfoNArgs (fn : Expr) (nargs : Nat) : MetaM FunInfo :=
  getFunInfoAux fn (some nargs)

end Lean.Meta
// ::::::::::::::
```

```
// GeneralizeTelescope.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.KAbstract

namespace Lean.Meta
namespace GeneralizeTelescope

structure Entry where
  expr     : Expr
  type     : Expr
  modified : Bool

partial def updateTypes (e eNew : Expr) (entries : Array Entry) (i : Nat) : MetaM (Array Entry) :=
  if h : i < entries.size then
    let entry := entries.get ⟨i, h⟩
    match entry with
    | ⟨_, type, _⟩ => do
      let typeAbst ← kabstract type e
      if typeAbst.hasLooseBVars then do
        let typeNew := typeAbst.instantiate1 eNew
        let entries := entries.set ⟨i, h⟩ { entry with type := typeNew, modified := true }
        updateTypes e eNew entries (i+1)
      else
        updateTypes e eNew entries (i+1)
  else
    pure entries

partial def generalizeTelescopeAux {α} (k : Array Expr → MetaM α)
    (entries : Array Entry) (i : Nat) (fvars : Array Expr) : MetaM α := do
  if h : i < entries.size then
    let replace (baseUserName : Name) (e : Expr) (type : Expr) : MetaM α := do
      let userName ← mkFreshUserName baseUserName
      withLocalDeclD userName type fun x => do
        let entries ← updateTypes e x entries (i+1)
        generalizeTelescopeAux k entries (i+1) (fvars.push x)
    match entries.get ⟨i, h⟩ with
    | ⟨e@(Expr.fvar fvarId _), type, false⟩ =>
      let localDecl ← getLocalDecl fvarId
      match localDecl with
      | LocalDecl.cdecl .. => generalizeTelescopeAux k entries (i+1) (fvars.push e)
      | LocalDecl.ldecl .. => replace localDecl.userName e type
    | ⟨e, type, modified⟩ =>
```

```
      if modified then
        unless (← isTypeCorrect type) do
          throwError! "failed to create telescope generalizing {entries.map Entry.expr}"
      replace `x e type
  else
    k fvars

end GeneralizeTelescope

open GeneralizeTelescope

/--
  Given expressions `es := #[e_1, e_2, ..., e_n]`, execute `k` with the
  free variables `(x_1 : A_1) (x_2 : A_2 [x_1]) ... (x_n : A_n [x_1, ... x_{n-1}])`.
  Moreover,
  - type of `e_1` is definitionally equal to `A_1`,
  - type of `e_2` is definitionally equal to `A_2[e_1]`.
  - ...
  - type of `e_n` is definitionally equal to `A_n[e_1, ..., e_{n-1}]`.

  This method tries to avoid the creation of new free variables. For example, if `e_i` is a
  free variable `x_i` and it is not a let-declaration variable, and its type does not depend on
  previous `e_j`s, the method will just use `x_i`.

  The telescope `x_1 ... x_n` can be used to create lambda and forall abstractions.
  Moreover, for any type correct lambda abstraction `f` constructed using `mkForall #[x_1, ..., x_n] ...`,
  The application `f e_1 ... e_n` is also type correct.

  The `kabstract` method is used to "locate" and abstract forward dependencies.
  That is, an occurrence of `e_i` in the of `e_j` for `j > i`.

  The method checks whether the abstract types `A_i` are type correct. Here is an example
  where `generalizeTelescope` fails to create the telescope `x_1 ... x_n`.
  Assume the local context contains `(n : Nat := 10) (xs : Vec Nat n) (ys : Vec Nat 10) (h : xs = ys)`.
  Then, assume we invoke `generalizeTelescope` with `es := #[10, xs, ys, h]`
  A type error is detected when processing `h`'s type. At this point, the method had successfully produced
  ```
    (x_1 : Nat) (xs : Vec Nat n) (x_2 : Vec Nat x_1)
  ```
  and the type for the new variable abstracting `h` is `xs = x_2` which is not type correct. -/
def generalizeTelescope {α} (es : Array Expr) (k : Array Expr → MetaM α) : MetaM α := do
  let es ← es.mapM fun e => do
    let type ← inferType e
    let type ← instantiateMVars type
    pure { expr := e, type := type, modified := false : Entry }
  generalizeTelescopeAux k es 0 #[]
```

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

namespace Lean.Meta

private def getDefInfo (info : ConstantInfo) : MetaM (Option ConstantInfo) := do
  match (← read).config.transparency with
  | TransparencyMode.all => return some info
  | TransparencyMode.default => return some info
  | m =>
    if (← isReducible info.name) then
      return some info
    else if m == TransparencyMode.instances && isGlobalInstance (← getEnv) info.name then
      return some info
    else
      return none

def getConst? (constName : Name) : MetaM (Option ConstantInfo) := do
  let env ← getEnv
  match env.find? constName with
  | some (info@(ConstantInfo.thmInfo _))  => getTheoremInfo info
  | some (info@(ConstantInfo.defnInfo _)) => getDefInfo info
  | some info                             => pure (some info)
  | none                                  => throwUnknownConstant constName

def getConstNoEx? (constName : Name) : MetaM (Option ConstantInfo) := do
  let env ← getEnv
  match env.find? constName with
  | some (info@(ConstantInfo.thmInfo _))  => getTheoremInfo info
  | some (info@(ConstantInfo.defnInfo _)) => getDefInfo info
  | some info                             => pure (some info)
  | none                                  => pure none

end Meta
// ::::::::::::::
// Inductive.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.ExprDefEq

/- Helper methods for inductive datatypes -/

namespace Lean.Meta

/- Return true if the types of the given constructors are compatible. -/
def compatibleCtors (ctorName₁ ctorName₂ : Name) : MetaM Bool := do
  let ctorInfo₁ ← getConstInfoCtor ctorName₁
  let ctorInfo₂ ← getConstInfoCtor ctorName₂
  if ctorInfo₁.induct != ctorInfo₂.induct then
    return false
  else
    let (_, _, ctorType₁) ← forallMetaTelescope ctorInfo₁.type
    let (_, _, ctorType₂) ← forallMetaTelescope ctorInfo₂.type
    isDefEq ctorType₁ ctorType₂

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

namespace Lean

/-
Auxiliary function for instantiating the loose bound variables in `e` with `args[start:stop]`.
This function is similar to `instantiateRevRange`, but it applies beta-reduction when
we instantiate a bound variable with a lambda expression.
Example: Given the term `#0 a`, and `start := 0, stop := 1, args := #[fun x => x]` the result is
`a` instead of `(fun x => x) a`.
This reduction is useful when we are inferring the type of eliminator-like applications.
For example, given `(n m : Nat) (f : Nat → Nat) (h : m = n)`,
the type of `Eq.subst (motive := fun x => f m = f x) h rfl`
is `motive n` which is `(fun (x : Nat) => f m = f x) n`
This function reduces the new application to `f m = f n`

We use it to implement `inferAppType`
```

```
-/
partial def Expr.instantiateBetaRevRange (e : Expr) (start : Nat) (stop : Nat) (args : Array Expr) : Expr :=
  if e.hasLooseBVars && stop > start then
    assert! stop ≤ args.size
    visit e 0 |>.run
  else
    e
where
  visit (e : Expr) (offset : Nat) : MonadStateCacheT (Expr × Nat) Expr Id Expr :=
    if offset >= e.looseBVarRange then
      -- `e` doesn't have free variables
      return e
    else checkCache (e, offset) fun _ => do
      match e with
      | Expr.forallE _ d b _   => return e.updateForallE! (← visit d offset) (← visit b (offset+1))
      | Expr.lam _ d b _       => return e.updateLambdaE! (← visit d offset) (← visit b (offset+1))
      | Expr.letE _ t v b _    => return e.updateLet! (← visit t offset) (← visit v offset) (← visit b (offset+1))
      | Expr.mdata _ b _       => return e.updateMData! (← visit b offset)
      | Expr.proj _ _ b _      => return e.updateProj! (← visit b offset)
      | Expr.app f a _         =>
        e.withAppRev fun f revArgs => do
        let fNew    ← visit f offset
        let revArgs ← revArgs.mapM (visit · offset)
        if f.isBVar then
          -- try to beta reduce if `f` was a bound variable
          return fNew.betaRev revArgs
        else
          return mkAppRev fNew revArgs
      | Expr.bvar vidx _       =>
        -- Recall that looseBVarRange for `Expr.bvar` is `vidx+1`.
        -- So, we must have offset ≤ vidx, since we are in the "else" branch of  `if offset >= e.looseBVarRange`
        let n := stop - start
        if vidx < offset + n then
          return args[stop - (vidx - offset) - 1].liftLooseBVars 0 offset
        else
          return mkBVar (vidx - n)
      -- The following cases are unreachable because they never contain loose bound variables
      | Expr.const .. => unreachable!
      | Expr.fvar ..  => unreachable!
      | Expr.mvar ..  => unreachable!
      | Expr.sort ..  => unreachable!
      | Expr.lit ..   => unreachable!

namespace Meta

def throwFunctionExpected {α} (f : Expr) : MetaM α :=
  throwError! "function expected{indentExpr f}"
```

```
private def inferAppType (f : Expr) (args : Array Expr) : MetaM Expr := do
  let mut fType ← inferType f
  let mut j := 0
  /- TODO: check whether `instantiateBetaRevRange` is too expensive, and
     use it only when `args` contains a lambda expression. -/
  for i in [:args.size] do
    match fType with
    | Expr.forallE _ _ b _ => fType := b
    | _ =>
      match (← whnf <| fType.instantiateBetaRevRange j i args) with
      | Expr.forallE _ _ b _ => j := i; fType := b
      | _ => throwFunctionExpected <| mkAppRange f 0 (i+1) args
  return fType.instantiateBetaRevRange j args.size args

def throwIncorrectNumberOfLevels {α} (constName : Name) (us : List Level) : MetaM α :=
  throwError! "incorrect number of universe levels {mkConst constName us}"

private def inferConstType (c : Name) (us : List Level) : MetaM Expr := do
  let cinfo ← getConstInfo c
  if cinfo.levelParams.length == us.length then
    return cinfo.instantiateTypeLevelParams us
  else
    throwIncorrectNumberOfLevels c us

private def inferProjType (structName : Name) (idx : Nat) (e : Expr) : MetaM Expr := do
  let failed {α} : Unit → MetaM α := fun _ =>
    throwError! "invalid projection{indentExpr (mkProj structName idx e)}"
  let structType ← inferType e
  let structType ← whnf structType
  matchConstStruct structType.getAppFn failed fun structVal structLvls ctorVal =>
    let n := structVal.numParams
    let structParams := structType.getAppArgs
    if n != structParams.size then failed ()
    else do
      let mut ctorType ← inferAppType (mkConst ctorVal.name structLvls) structParams
      for i in [:idx] do
        ctorType ← whnf ctorType
        match ctorType with
        | Expr.forallE _ _ body _ =>
          if body.hasLooseBVars then
            ctorType := body.instantiate1 $ mkProj structName i e
          else
            ctorType := body
        | _ => failed ()
      ctorType ← whnf ctorType
      match ctorType with
```

```
      | Expr.forallE _ d _ _ => pure d
      | _                    => failed ()

def throwTypeExcepted {α} (type : Expr) : MetaM α :=
  throwError! "type expected{indentExpr type}"

def getLevel (type : Expr) : MetaM Level := do
  let typeType ← inferType type
  let typeType ← whnfD typeType
  match typeType with
  | Expr.sort lvl _    => pure lvl
  | Expr.mvar mvarId _ =>
    if (← isReadOnlyOrSyntheticOpaqueExprMVar mvarId) then
      throwTypeExcepted type
    else
      let lvl ← mkFreshLevelMVar
      assignExprMVar mvarId (mkSort lvl)
      pure lvl
  | _ => throwTypeExcepted type

private def inferForallType (e : Expr) : MetaM Expr :=
  forallTelescope e fun xs e => do
    let lvl  ← getLevel e
    let lvl  ← xs.foldrM (init := lvl) fun x lvl => do
      let xType    ← inferType x
      let xTypeLvl ← getLevel xType
      pure $ mkLevelIMax' xTypeLvl lvl
    pure $ mkSort lvl.normalize

/- Infer type of lambda and let expressions -/
private def inferLambdaType (e : Expr) : MetaM Expr :=
  lambdaLetTelescope e fun xs e => do
    let type ← inferType e
    mkForallFVars xs type

@[inline] private def withLocalDecl' {α} (name : Name) (bi : BinderInfo) (type : Expr) (x : Expr → MetaM α) : MetaM α :=
  savingCache do
    let fvarId ← mkFreshId
    withReader (fun ctx => { ctx with lctx := ctx.lctx.mkLocalDecl fvarId name type bi }) do
      x (mkFVar fvarId)

def throwUnknownMVar {α} (mvarId : MVarId) : MetaM α :=
  throwError! "unknown metavariable '{mkMVar mvarId}'"

private def inferMVarType (mvarId : MVarId) : MetaM Expr := do
  match (← getMCtx).findDecl? mvarId with
  | some d => pure d.type
```

```
  | none   => throwUnknownMVar mvarId

private def inferFVarType (fvarId : FVarId) : MetaM Expr := do
  match (← getLCtx).find? fvarId with
  | some d => pure d.type
  | none   => throwUnknownFVar fvarId

@[inline] private def checkInferTypeCache (e : Expr) (inferType : MetaM Expr) : MetaM Expr := do
  match (← get).cache.inferType.find? e with
  | some type => pure type
  | none =>
    let type ← inferType
    modifyInferTypeCache fun c => c.insert e type
    pure type

def inferTypeImp (e : Expr) : MetaM Expr :=
  let rec infer : Expr → MetaM Expr
    | Expr.const c [] _        => inferConstType c []
    | Expr.const c us _        => checkInferTypeCache e (inferConstType c us)
    | e@(Expr.proj n i s _)    => checkInferTypeCache e (inferProjType n i s)
    | e@(Expr.app f _ _)       => checkInferTypeCache e (inferAppType f.getAppFn e.getAppArgs)
    | Expr.mvar mvarId _       => inferMVarType mvarId
    | Expr.fvar fvarId _       => inferFVarType fvarId
    | Expr.bvar bidx _         => throwError! "unexpected bound variable {mkBVar bidx}"
    | Expr.mdata _ e _         => infer e
    | Expr.lit v _             => pure v.type
    | Expr.sort lvl _          => pure $ mkSort (mkLevelSucc lvl)
    | e@(Expr.forallE _ _ _ _) => checkInferTypeCache e (inferForallType e)
    | e@(Expr.lam _ _ _ _)     => checkInferTypeCache e (inferLambdaType e)
    | e@(Expr.letE _ _ _ _ _)  => checkInferTypeCache e (inferLambdaType e)
  withTransparency TransparencyMode.default (infer e)

@[builtinInit] def setInferTypeRef : IO Unit :=
inferTypeRef.set inferTypeImp

/--
  Return `LBool.true` if given level is always equivalent to universe level zero.
  It is used to implement `isProp`. -/
private def isAlwaysZero : Level → Bool
  | Level.zero _     => true
  | Level.mvar _ _   => false
  | Level.param _ _  => false
  | Level.succ _ _   => false
  | Level.max u v _  => isAlwaysZero u && isAlwaysZero v
  | Level.imax _ u _ => isAlwaysZero u

/--
```

```
  `isArrowProp type n` is an "approximate" predicate which returns `LBool.true`
   if `type` is of the form `A_1 -> ... -> A_n -> Prop`.
   Remark: `type` can be a dependent arrow. -/
private partial def isArrowProp : Expr → Nat → MetaM LBool
  | Expr.sort u _,        0   => return isAlwaysZero (← instantiateLevelMVars u) |>.toLBool
  | Expr.forallE _ _ _ _, 0   => pure LBool.false
  | Expr.forallE _ _ b _, n+1 => isArrowProp b n
  | Expr.letE _ _ _ b _,  n   => isArrowProp b n
  | Expr.mdata _ e _,     n   => isArrowProp e n
  | _,                    _   => pure LBool.undef

/--
  `isPropQuickApp f n` is an "approximate" predicate which returns `LBool.true`
   if `f` applied to `n` arguments is a proposition. -/
private partial def isPropQuickApp : Expr → Nat → MetaM LBool
  | Expr.const c lvls _, arity   => do let constType ← inferConstType c lvls; isArrowProp constType arity
  | Expr.fvar fvarId _,  arity   => do let fvarType  ← inferFVarType fvarId;  isArrowProp fvarType arity
  | Expr.mvar mvarId _,  arity   => do let mvarType  ← inferMVarType mvarId;  isArrowProp mvarType arity
  | Expr.app f _ _,      arity   => isPropQuickApp f (arity+1)
  | Expr.mdata _ e _,    arity   => isPropQuickApp e arity
  | Expr.letE _ _ _ b _, arity   => isPropQuickApp b arity
  | Expr.lam _ _ _ _,    0       => pure LBool.false
  | Expr.lam _ _ b _,    arity+1 => isPropQuickApp b arity
  | _,                   _       => pure LBool.undef

/--
  `isPropQuick e` is an "approximate" predicate which returns `LBool.true`
  if `e` is a proposition. -/
partial def isPropQuick : Expr → MetaM LBool
  | Expr.bvar _ _         => pure LBool.undef
  | Expr.lit _ _          => pure LBool.false
  | Expr.sort _ _         => pure LBool.false
  | Expr.lam _ _ _ _      => pure LBool.false
  | Expr.letE _ _ _ b _   => isPropQuick b
  | Expr.proj _ _ _ _     => pure LBool.undef
  | Expr.forallE _ _ b _  => isPropQuick b
  | Expr.mdata _ e _      => isPropQuick e
  | Expr.const c lvls _   => do let constType ← inferConstType c lvls; isArrowProp constType 0
  | Expr.fvar fvarId _    => do let fvarType  ← inferFVarType fvarId;  isArrowProp fvarType 0
  | Expr.mvar mvarId _    => do let mvarType  ← inferMVarType mvarId;  isArrowProp mvarType 0
  | Expr.app f _ _        => isPropQuickApp f 1

/-- `isProp whnf e` return `true` if `e` is a proposition.

     If `e` contains metavariables, it may not be possible
     to decide whether is a proposition or not. We return `false` in this
     case. We considered using `LBool` and retuning `LBool.undef`, but
```

```
     we have no applications for it. -/
def isProp (e : Expr) : MetaM Bool := do
  let r ← isPropQuick e
  match r with
  | LBool.true  => pure true
  | LBool.false => pure false
  | LBool.undef =>
    let type ← inferType e
    let type ← whnfD type
    match type with
    | Expr.sort u _ => return isAlwaysZero (← instantiateLevelMVars u)
    | _             => pure false

/--
  `isArrowProposition type n` is an "approximate" predicate which returns `LBool.true`
   if `type` is of the form `A_1 -> ... -> A_n -> B`, where `B` is a proposition.
   Remark: `type` can be a dependent arrow. -/
private partial def isArrowProposition : Expr → Nat → MetaM LBool
  | Expr.forallE _ _ b _, n+1 => isArrowProposition b n
  | Expr.letE _ _ _ b _,  n   => isArrowProposition b n
  | Expr.mdata _ e _,     n   => isArrowProposition e n
  | type,                 0   => isPropQuick type
  | _,                    _   => pure LBool.undef

mutual
/--
  `isProofQuickApp f n` is an "approximate" predicate which returns `LBool.true`
   if `f` applied to `n` arguments is a proof. -/
private partial def isProofQuickApp : Expr → Nat → MetaM LBool
  | Expr.const c lvls _, arity   => do let constType ← inferConstType c lvls; isArrowProposition constType arity
  | Expr.fvar fvarId _,  arity   => do let fvarType  ← inferFVarType fvarId;  isArrowProposition fvarType arity
  | Expr.mvar mvarId _,  arity   => do let mvarType  ← inferMVarType mvarId;  isArrowProposition mvarType arity
  | Expr.app f _ _,      arity   => isProofQuickApp f (arity+1)
  | Expr.mdata _ e _,    arity   => isProofQuickApp e arity
  | Expr.letE _ _ _ b _, arity   => isProofQuickApp b arity
  | Expr.lam _ _ b _,    0       => isProofQuick b
  | Expr.lam _ _ b _,    arity+1 => isProofQuickApp b arity
  | _,                   _       => pure LBool.undef

/--
  `isProofQuick e` is an "approximate" predicate which returns `LBool.true`
  if `e` is a proof. -/
partial def isProofQuick : Expr → MetaM LBool
  | Expr.bvar _ _         => pure LBool.undef
  | Expr.lit _ _          => pure LBool.false
  | Expr.sort _ _         => pure LBool.false
  | Expr.lam _ _ b _      => isProofQuick b
```

```
  | Expr.letE _ _ _ b _   => isProofQuick b
  | Expr.proj _ _ _ _     => pure LBool.undef
  | Expr.forallE _ _ b _  => pure LBool.false
  | Expr.mdata _ e _      => isProofQuick e
  | Expr.const c lvls _   => do let constType ← inferConstType c lvls; isArrowProposition constType 0
  | Expr.fvar fvarId _    => do let fvarType  ← inferFVarType fvarId;  isArrowProposition fvarType 0
  | Expr.mvar mvarId _    => do let mvarType  ← inferMVarType mvarId;  isArrowProposition mvarType 0
  | Expr.app f _ _        => isProofQuickApp f 1

end

def isProof (e : Expr) : MetaM Bool := do
  let r ← isProofQuick e
  match r with
  | LBool.true  => pure true
  | LBool.false => pure false
  | LBool.undef => do
    let type ← inferType e
    Meta.isProp type

/--
  `isArrowType type n` is an "approximate" predicate which returns `LBool.true`
   if `type` is of the form `A_1 -> ... -> A_n -> Sort _`.
   Remark: `type` can be a dependent arrow. -/
private partial def isArrowType : Expr → Nat → MetaM LBool
  | Expr.sort u _,        0   => pure LBool.true
  | Expr.forallE _ _ _ _, 0   => pure LBool.false
  | Expr.forallE _ _ b _, n+1 => isArrowType b n
  | Expr.letE _ _ _ b _,  n   => isArrowType b n
  | Expr.mdata _ e _,     n   => isArrowType e n
  | _,                    _   => pure LBool.undef

/--
  `isTypeQuickApp f n` is an "approximate" predicate which returns `LBool.true`
   if `f` applied to `n` arguments is a type. -/
private partial def isTypeQuickApp : Expr → Nat → MetaM LBool
  | Expr.const c lvls _, arity   => do let constType ← inferConstType c lvls; isArrowType constType arity
  | Expr.fvar fvarId _,  arity   => do let fvarType  ← inferFVarType fvarId;  isArrowType fvarType arity
  | Expr.mvar mvarId _,  arity   => do let mvarType  ← inferMVarType mvarId;  isArrowType mvarType arity
  | Expr.app f _ _,      arity   => isTypeQuickApp f (arity+1)
  | Expr.mdata _ e _,    arity   => isTypeQuickApp e arity
  | Expr.letE _ _ _ b _, arity   => isTypeQuickApp b arity
  | Expr.lam _ _ _ _,    0       => pure LBool.false
  | Expr.lam _ _ b _,    arity+1 => isTypeQuickApp b arity
  | _,                   _       => pure LBool.undef

/--
```

```
  `isTypeQuick e` is an "approximate" predicate which returns `LBool.true`
  if `e` is a type. -/
partial def isTypeQuick : Expr → MetaM LBool
  | Expr.bvar _ _         => pure LBool.undef
  | Expr.lit _ _          => pure LBool.false
  | Expr.sort _ _         => pure LBool.true
  | Expr.lam _ _ _ _      => pure LBool.false
  | Expr.letE _ _ _ b _   => isTypeQuick b
  | Expr.proj _ _ _ _     => pure LBool.undef
  | Expr.forallE _ _ b _  => pure LBool.true
  | Expr.mdata _ e _      => isTypeQuick e
  | Expr.const c lvls _   => do let constType ← inferConstType c lvls; isArrowType constType 0
  | Expr.fvar fvarId _    => do let fvarType  ← inferFVarType fvarId;  isArrowType fvarType 0
  | Expr.mvar mvarId _    => do let mvarType  ← inferMVarType mvarId;  isArrowType mvarType 0
  | Expr.app f _ _        => isTypeQuickApp f 1

def isType (e : Expr) : MetaM Bool := do
  let r ← isTypeQuick e
  match r with
  | LBool.true  => pure true
  | LBool.false => pure false
  | LBool.undef =>
    let type ← inferType e
    let type ← whnfD type
    match type with
    | Expr.sort _ _ => pure true
    | _             => pure false

partial def isTypeFormerType (type : Expr) : MetaM Bool := do
  let type ← whnfD type
  match type with
  | Expr.sort _ _ => pure true
  | Expr.forallE n d b c =>
    withLocalDecl' n c.binderInfo d fun fvar =>
    isTypeFormerType (b.instantiate1 fvar)
  | _ => pure false

/--
  Return true iff `e : Sort _` or `e : (forall As, Sort _)`.
  Remark: it subsumes `isType` -/
def isTypeFormer (e : Expr) : MetaM Bool := do
  let type ← inferType e
  isTypeFormerType type

end Lean.Meta
// ::::::::::::
// Instances.lean
```

```
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.ScopedEnvExtension
import Lean.Meta.DiscrTree

namespace Lean.Meta

structure InstanceEntry where
  keys        : Array DiscrTree.Key
  val         : Expr
  priority    : Nat
  globalName? : Option Name := none
  deriving Inhabited

instance : BEq InstanceEntry where
  beq e₁ e₂ := e₁.val == e₂.val

instance : ToFormat InstanceEntry where
  format e := match e.globalName? with
    | some n => fmt n
    | _      => "<local>"

structure Instances where
  discrTree       : DiscrTree InstanceEntry := DiscrTree.empty
  globalInstances : NameSet := {}
  deriving Inhabited

def addInstanceEntry (d : Instances) (e : InstanceEntry) : Instances := {
  d with
    discrTree := d.discrTree.insertCore e.keys e
    globalInstances := match e.globalName? with
      | some n => d.globalInstances.insert n
      | none   => d.globalInstances
}

builtin_initialize instanceExtension : SimpleScopedEnvExtension InstanceEntry Instances ←
  registerSimpleScopedEnvExtension {
    name     := `instanceExt
    initial  := {}
    addEntry := addInstanceEntry
  }

private def mkInstanceKey (e : Expr) : MetaM (Array DiscrTree.Key) := do
```

```
  let type ← inferType e
  withNewMCtxDepth do
    let (_, _, type) ← forallMetaTelescopeReducing type
    DiscrTree.mkPath type

def addInstance (declName : Name) (attrKind : AttributeKind) (prio : Nat) : MetaM Unit := do
  let cinfo ← getConstInfo declName
  let c := mkConst declName (cinfo.levelParams.map mkLevelParam)
  let keys ← mkInstanceKey c
  instanceExtension.add { keys := keys, val := c, priority := prio, globalName? := declName } attrKind

builtin_initialize
  registerBuiltinAttribute {
    name  := `instance
    descr := "type class instance"
    add   := fun declName stx attrKind => do
      let prio ← getAttrParamOptPrio stx[1]
      discard <| addInstance declName attrKind prio |>.run {} {}
  }

@[export lean_is_instance]
def isGlobalInstance (env : Environment) (declName : Name) : Bool :=
  Meta.instanceExtension.getState env |>.globalInstances.contains declName

def getGlobalInstancesIndex : MetaM (DiscrTree InstanceEntry) :=
  return Meta.instanceExtension.getState (← getEnv) |>.discrTree

/- Default instance support -/

structure DefaultInstanceEntry where
  className    : Name
  instanceName : Name
  priority     : Nat

abbrev PrioritySet := Std.RBTree Nat (.>.)

structure DefaultInstances where
  defaultInstances : NameMap (List (Name × Nat)) := {}
  priorities       : PrioritySet := {}
  deriving Inhabited

def addDefaultInstanceEntry (d : DefaultInstances) (e : DefaultInstanceEntry) : DefaultInstances :=
  let d := { d with priorities := d.priorities.insert e.priority }
  match d.defaultInstances.find? e.className with
  | some insts => { d with defaultInstances := d.defaultInstances.insert e.className <| (e.instanceName, e.priority) :: insts }
  | none       => { d with defaultInstances := d.defaultInstances.insert e.className [(e.instanceName, e.priority)] }

```

```
builtin_initialize defaultInstanceExtension : SimplePersistentEnvExtension DefaultInstanceEntry DefaultInstances ←
  registerSimplePersistentEnvExtension {
    name          := `defaultInstanceExt
    addEntryFn    := addDefaultInstanceEntry
    addImportedFn := fun es => (mkStateFromImportedEntries addDefaultInstanceEntry {} es)
  }

def addDefaultInstance (declName : Name) (prio : Nat := 0) : MetaM Unit := do
  match (← getEnv).find? declName with
  | none => throwError! "unknown constant '{declName}'"
  | some info =>
    forallTelescopeReducing info.type fun _ type => do
      match type.getAppFn with
      | Expr.const className _ _ =>
        unless isClass (← getEnv) className do
          throwError! "invalid default instance '{declName}', it has type '({className} ...)', but {className}' is not a type class"
        setEnv <| defaultInstanceExtension.addEntry (← getEnv) { className := className, instanceName := declName, priority := prio }
      | _ => throwError! "invalid default instance '{declName}', type must be of the form '(C ...)' where 'C' is a type class"

builtin_initialize
  registerBuiltinAttribute {
    name  := `defaultInstance
    descr := "type class default instance"
    add   := fun declName stx kind => do
      let prio ← getAttrParamOptPrio stx[1]
      unless kind == AttributeKind.global do throwError "invalid attribute 'defaultInstance', must be global"
      discard <| addDefaultInstance declName prio |>.run {} {}
  }

def getDefaultInstancesPriorities [Monad m] [MonadEnv m] : m PrioritySet :=
  return defaultInstanceExtension.getState (← getEnv) |>.priorities

def getDefaultInstances [Monad m] [MonadEnv m] (className : Name) : m (List (Name × Nat)) :=
  return defaultInstanceExtension.getState (← getEnv) |>.defaultInstances.find? className |>.getD []

end Lean.Meta
// ::::::::::::::
// KAbstract.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: Leonardo de Moura
-/
import Lean.Data.Occurrences
import Lean.HeadIndex
import Lean.Meta.ExprDefEq
```

```
namespace Lean.Meta

def kabstract (e : Expr) (p : Expr) (occs : Occurrences := Occurrences.all) : MetaM Expr := do
  let e ← instantiateMVars e
  if p.isFVar && occs == Occurrences.all then
    return e.abstract #[p] -- Easy case
  else
    let pHeadIdx := p.toHeadIndex
    let pNumArgs := p.headNumArgs
    let rec visit (e : Expr) (offset : Nat) : StateRefT Nat MetaM Expr := do
      let visitChildren : Unit → StateRefT Nat MetaM Expr := fun _ => do
        match e with
        | Expr.app f a _       => return e.updateApp! (← visit f offset) (← visit a offset)
        | Expr.mdata _ b _     => return e.updateMData! (← visit b offset)
        | Expr.proj _ _ b _    => return e.updateProj! (← visit b offset)
        | Expr.letE _ t v b _  => return e.updateLet! (← visit t offset) (← visit v offset) (← visit b (offset+1))
        | Expr.lam _ d b _     => return e.updateLambdaE! (← visit d offset) (← visit b (offset+1))
        | Expr.forallE _ d b _ => return e.updateForallE! (← visit d offset) (← visit b (offset+1))
        | e                    => return e
      if e.hasLooseBVars then
        visitChildren ()
      else if e.toHeadIndex != pHeadIdx || e.headNumArgs != pNumArgs then
        visitChildren ()
      else if (← isDefEq e p) then
        let i ← get
        set (i+1)
        if occs.contains i then
          pure (mkBVar offset)
        else
          visitChildren ()
      else
        visitChildren ()
    visit e 0 |>.run' 1

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

```

```
namespace Lean.Meta

private partial def decAux? : Level → MetaM (Option Level)
  | Level.zero _        => return none
  | Level.param _ _     => return none
  | Level.mvar mvarId _ => do
    let mctx ← getMCtx
    match mctx.getLevelAssignment? mvarId with
    | some u => decAux? u
    | none   =>
      if (← isReadOnlyLevelMVar mvarId) then
        return none
      else
        let u ← mkFreshLevelMVar
        assignLevelMVar mvarId (mkLevelSucc u)
        return u
  | Level.succ u _  => return u
  | u =>
    let process (u v : Level) : MetaM (Option Level) := do
      match (← decAux? u) with
      | none   => return none
      | some u => do
        match (← decAux? v) with
        | none   => return none
        | some v => return mkLevelMax' u v
    match u with
    | Level.max u v _  => process u v
    /- Remark: If `decAux? v` returns `some ...`, then `imax u v` is equivalent to `max u v`. -/
    | Level.imax u v _ => process u v
    | _                => unreachable!

def decLevel? (u : Level) : MetaM (Option Level) := do
  let mctx ← getMCtx
  match (← decAux? u) with
  | some v => return some v
  | none   => do
    modify fun s => { s with mctx := mctx }
    return none

def decLevel (u : Level) : MetaM Level := do
  match (← decLevel? u) with
  | some u => return u
  | none   => throwError! "invalid universe level, {u} is not greater than 0"

/- This method is useful for inferring universe level parameters for function that take arguments such as `{α : Type u}`.
   Recall that `Type u` is `Sort (u+1)` in Lean. Thus, given `α`, we must infer its universe level,
   and then decrement 1 to obtain `u`. -/
```

```
def getDecLevel (type : Expr) : MetaM Level := do
  decLevel (← getLevel type)

/--
  Return true iff `lvl` occurs in `max u_1 ... u_n` and `lvl != u_i` for all `i in [1, n]`.
  That is, `lvl` is a proper level subterm of some `u_i`. -/
private def strictOccursMax (lvl : Level) : Level → Bool
  | Level.max u v _ => visit u || visit v
  | _               => false
where
  visit : Level → Bool
    | Level.max u v _ => visit u || visit v
    | u               => u != lvl && lvl.occurs u

/-- `mkMaxArgsDiff mvarId (max u_1 ... (mvar mvarId) ... u_n) v` => `max v u_1 ... u_n` -/
private def mkMaxArgsDiff (mvarId : MVarId) : Level → Level → Level
  | Level.max u v _,     acc => mkMaxArgsDiff mvarId v <| mkMaxArgsDiff mvarId u acc
  | l@(Level.mvar id _), acc => if id != mvarId then mkLevelMax' acc l else acc
  | l,                   acc => mkLevelMax' acc l

/--
  Solve `?m =?= max ?m v` by creating a fresh metavariable `?n`
  and assigning `?m := max ?n v` -/
private def solveSelfMax (mvarId : MVarId) (v : Level) : MetaM Unit := do
  assert! v.isMax
  let n ← mkFreshLevelMVar
  assignLevelMVar mvarId <| mkMaxArgsDiff mvarId v n

private def postponeIsLevelDefEq (lhs : Level) (rhs : Level) : MetaM Unit :=
  modifyPostponed fun postponed => postponed.push { lhs := lhs, rhs := rhs }

mutual

  private partial def solve (u v : Level) : MetaM LBool := do
    match u, v with
    | Level.mvar mvarId _, _ =>
      if (← isReadOnlyLevelMVar mvarId) then
        return LBool.undef
      else if !u.occurs v then
        assignLevelMVar u.mvarId! v
        return LBool.true
      else if v.isMax && !strictOccursMax u v then
        solveSelfMax u.mvarId! v
        return LBool.true
      else
        return LBool.undef
    | Level.zero _, Level.max v₁ v₂ _ =>
```

```
      Bool.toLBool <$> (isLevelDefEqAux levelZero v₁ <&&> isLevelDefEqAux levelZero v₂)
    | Level.zero _, Level.imax _ v₂ _ =>
      Bool.toLBool <$> isLevelDefEqAux levelZero v₂
    | Level.zero _, Level.succ .. => return LBool.false
    | Level.succ u _, v =>
      if u.isMVar && u.occurs v then
        return LBool.undef
      else
        match (← Meta.decLevel? v) with
        | some v => Bool.toLBool <$> isLevelDefEqAux u v
        | none   => return LBool.undef
    | _, _ => return LBool.undef

  partial def isLevelDefEqAux : Level → Level → MetaM Bool
    | Level.succ lhs _, Level.succ rhs _ => isLevelDefEqAux lhs rhs
    | lhs, rhs => do
      if lhs == rhs then
        return true
      else
        trace[Meta.isLevelDefEq.step]! "{lhs} =?= {rhs}"
        let lhs' ← instantiateLevelMVars lhs
        let lhs' := lhs'.normalize
        let rhs' ← instantiateLevelMVars rhs
        let rhs' := rhs'.normalize
        if lhs != lhs' || rhs != rhs' then
          isLevelDefEqAux lhs' rhs'
        else
          let r ← solve lhs rhs;
          if r != LBool.undef then
            return r == LBool.true
          else
            let r ← solve rhs lhs;
            if r != LBool.undef then
              return r == LBool.true
            else do
              let mctx ← getMCtx
              if !mctx.hasAssignableLevelMVar lhs && !mctx.hasAssignableLevelMVar rhs then
                let ctx ← read
                if ctx.config.isDefEqStuckEx && (lhs.isMVar || rhs.isMVar) then do
                  trace[Meta.isLevelDefEq.stuck]! "{lhs} =?= {rhs}"
                  Meta.throwIsDefEqStuck
                else
                  return false
              else
                postponeIsLevelDefEq lhs rhs
                return true
end
```

```
def isListLevelDefEqAux : List Level → List Level → MetaM Bool
  | [],    []    => return true
  | u::us, v::vs => isLevelDefEqAux u v <&&> isListLevelDefEqAux us vs
  | _,     _     => return false

private def getNumPostponed : MetaM Nat := do
  return (← getPostponed).size

open Std (PersistentArray)

private def getResetPostponed : MetaM (PersistentArray PostponedEntry) := do
  let ps ← getPostponed
  setPostponed {}
  return ps

private def processPostponedStep : MetaM Bool :=
  traceCtx `Meta.isLevelDefEq.postponed.step do
    let ps ← getResetPostponed
    for p in ps do
      unless (← isLevelDefEqAux p.lhs p.rhs) do
        return false
    return true

private partial def processPostponed (mayPostpone : Bool := true) : MetaM Bool := do
if (← getNumPostponed) == 0 then
  return true
else
  traceCtx `Meta.isLevelDefEq.postponed do
    let rec loop : MetaM Bool := do
      let numPostponed ← getNumPostponed
      if numPostponed == 0 then
        return true
      else
        trace[Meta.isLevelDefEq.postponed]! "processing #{numPostponed} postponed is-def-eq level constraints"
        if !(← processPostponedStep) then
          return false
        else
          let numPostponed' ← getNumPostponed
          if numPostponed' == 0 then
            return true
          else if numPostponed' < numPostponed then
            loop
          else
            trace[Meta.isLevelDefEq.postponed]! "no progress solving pending is-def-eq level constraints"
            return mayPostpone
    loop
```

```
private def restore (env : Environment) (mctx : MetavarContext) (postponed : PersistentArray PostponedEntry) : MetaM Unit := do
  setEnv env
  setMCtx mctx
  setPostponed postponed

/--
  `commitWhen x` executes `x` and process all postponed universe level constraints produced by `x`.
  We keep the modifications only if `processPostponed` return true and `x` returned `true`.

  Remark: postponed universe level constraints must be solved before returning. Otherwise,
  we don't know whether `x` really succeeded. -/
@[specialize] def commitWhen (x : MetaM Bool) (mayPostpone : Bool := true) : MetaM Bool := do
  let env  ← getEnv
  let mctx ← getMCtx
  let postponed ← getResetPostponed
  try
    if (← x) then
      if (← processPostponed mayPostpone) then
        return true
      else
        restore env mctx postponed
        return false
    else
      restore env mctx postponed
      return false
  catch ex =>
    restore env mctx postponed
    throw ex

private def postponedToMessageData (ps : PersistentArray PostponedEntry) : MessageData := do
  let mut r := MessageData.nil
  for p in ps do
    r := m!"{r}\n{p.lhs} =?= {p.rhs}"
  return r

@[specialize] def withoutPostponingUniverseConstraintsImp {α} (x : MetaM α) : MetaM α := do
  let postponed ← getResetPostponed
  try
    let a ← x
    unless (← processPostponed (mayPostpone := false)) do
      throwError! "stuck at solving universe constraints{MessageData.nestD (postponedToMessageData (← getPostponed))}"
    setPostponed postponed
    return a
  catch ex =>
    setPostponed postponed
    throw ex
```

```
@[inline] def withoutPostponingUniverseConstraints {α m} [MonadControlT MetaM m] [Monad m] : m α → m α :=
  mapMetaM <| withoutPostponingUniverseConstraintsImp

def isLevelDefEq (u v : Level) : MetaM Bool :=
  traceCtx `Meta.isLevelDefEq do
    let b ← commitWhen (mayPostpone := true) <| Meta.isLevelDefEqAux u v
    trace[Meta.isLevelDefEq]! "{u} =?= {v} ... {if b then "success" else "failure"}"
    return b

def isExprDefEq (t s : Expr) : MetaM Bool :=
  traceCtx `Meta.isDefEq do
    let b ← commitWhen (mayPostpone := true) <| Meta.isExprDefEqAux t s
    trace[Meta.isDefEq]! "{t} =?= {s} ... {if b then "success" else "failure"}"
    return b

abbrev isDefEq (t s : Expr) : MetaM Bool :=
  isExprDefEq t s

def isExprDefEqGuarded (a b : Expr) : MetaM Bool := do
  try isExprDefEq a b catch _ => return false

abbrev isDefEqGuarded (t s : Expr) : MetaM Bool :=
  isExprDefEqGuarded t s

def isDefEqNoConstantApprox (t s : Expr) : MetaM Bool :=
  approxDefEq <| isDefEq t s

builtin_initialize
  registerTraceClass `Meta.isLevelDefEq
  registerTraceClass `Meta.isLevelDefEq.step
  registerTraceClass `Meta.isLevelDefEq.postponed

end Lean.Meta
// ::::::::::::::
// Match.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.Match.MatchPatternAttr
import Lean.Meta.Match.Match
import Lean.Meta.Match.CaseValues
import Lean.Meta.Match.CaseArraySizes
```

```
namespace Lean

builtin_initialize registerTraceClass `Meta.Match

end Lean
// ::::::::::::::
// MatchUtil.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Util.Recognizers
import Lean.Meta.Basic

namespace Lean.Meta

@[inline] def testHelper (e : Expr) (p : Expr → MetaM Bool) : MetaM Bool := do
  if (← p e) then
    return true
  else
    p (← whnf e)

@[inline] def matchHelper? (e : Expr) (p? : Expr → MetaM (Option α)) : MetaM (Option α) := do
  match (← p? e) with
  | none => p? (← whnf e)
  | s    => return s

def matchEq? (e : Expr) : MetaM (Option (Expr × Expr × Expr)) :=
  matchHelper? e fun e => return Expr.eq? e

def matchFalse (e : Expr) : MetaM Bool := do
  testHelper e fun e => return e.isConstOf ``False

def matchNot? (e : Expr) : MetaM (Option Expr) :=
  matchHelper? e fun e => do
    if let some e := e.not? then
      return e
    else if let some (a, b) := e.arrow? then
      if (← matchFalse b) then return some a else return none
    else
      return none

def matchNe? (e : Expr) : MetaM (Option (Expr × Expr × Expr)) :=
  matchHelper? e fun e => do
    if let some r := e.ne? then
```

```
      return r
    else if let some e ← matchNot? e then
      matchEq? e
    else
      return none

def matchConstructorApp? (e : Expr) : MetaM (Option ConstructorVal) := do
  let env ← getEnv
  matchHelper? e fun e =>
    return e.isConstructorApp? env

end Lean.Meta
// ::::::::::::::
// Offset.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Data.LBool
import Lean.Meta.InferType

namespace Lean.Meta

private abbrev withInstantiatedMVars (e : Expr) (k : Expr → OptionT MetaM α) : OptionT MetaM α := do
  let eNew ← instantiateMVars e
  if eNew.getAppFn.isMVar then
    failure
  else
    k eNew

/--
  Evaluate simple `Nat` expressions.
  Remark: this method assumes the given expression has type `Nat`. -/
partial def evalNat : Expr → OptionT MetaM Nat
  | Expr.lit (Literal.natVal n) _ => return n
  | Expr.mdata _ e _              => evalNat e
  | Expr.const `Nat.zero ..       => return 0
  | e@(Expr.app ..)               => visit e
  | e@(Expr.mvar ..)              => visit e
  | _                             => failure
where
  visit e := do
    let f := e.getAppFn
    match f with
    | Expr.mvar .. => withInstantiatedMVars e evalNat
```

```
  | Expr.const c _ _ =>
    let nargs := e.getAppNumArgs
    if c == ``Nat.succ && nargs == 1 then
      let v ← evalNat (e.getArg! 0)
      return v+1
    else if c == ``Nat.add && nargs == 2 then
      let v₁ ← evalNat (e.getArg! 0)
      let v₂ ← evalNat (e.getArg! 1)
      return v₁ + v₂
    else if c == ``Nat.sub && nargs == 2 then
      let v₁ ← evalNat (e.getArg! 0)
      let v₂ ← evalNat (e.getArg! 1)
      return v₁ - v₂
    else if c == ``Nat.mul && nargs == 2 then
      let v₁ ← evalNat (e.getArg! 0)
      let v₂ ← evalNat (e.getArg! 1)
      return v₁ * v₂
    else if c == ``Add.add && nargs == 4 then
      let v₁ ← evalNat (e.getArg! 2)
      let v₂ ← evalNat (e.getArg! 3)
      return v₁ + v₂
    else if c == ``Sub.sub && nargs == 4 then
      let v₁ ← evalNat (e.getArg! 2)
      let v₂ ← evalNat (e.getArg! 3)
      return v₁ - v₂
    else if c == ``Mul.mul && nargs == 4 then
      let v₁ ← evalNat (e.getArg! 2)
      let v₂ ← evalNat (e.getArg! 3)
      return v₁ * v₂
    else if c == ``HAdd.hAdd && nargs == 6 then
      let v₁ ← evalNat (e.getArg! 3)
      let v₂ ← evalNat (e.getArg! 5)
      return v₁ + v₂
    else if c == ``HSub.hSub && nargs == 6 then
      let v₁ ← evalNat (e.getArg! 4)
      let v₂ ← evalNat (e.getArg! 5)
      return v₁ - v₂
    else if c == ``HMul.hMul && nargs == 6 then
      let v₁ ← evalNat (e.getArg! 4)
      let v₂ ← evalNat (e.getArg! 5)
      return v₁ * v₂
    else if c == ``OfNat.ofNat && nargs == 3 then
      evalNat (e.getArg! 1)
    else
      failure
  | _ => failure
```

```
/- Quick function for converting `e` into `s + k` s.t. `e` is definitionally equal to `Nat.add s k`. -/
private partial def getOffsetAux : Expr → Bool → OptionT MetaM (Expr × Nat)
  | e@(Expr.app _ a _), top => do
    let f := e.getAppFn
    match f with
    | Expr.mvar .. => withInstantiatedMVars e (getOffsetAux · top)
    | Expr.const c _ _ =>
      let nargs := e.getAppNumArgs
      if c == ``Nat.succ && nargs == 1 then do
        let (s, k) ← getOffsetAux a false
        pure (s, k+1)
      else if c == ``Nat.add && nargs == 2 then do
        let v      ← evalNat (e.getArg! 1)
        let (s, k) ← getOffsetAux (e.getArg! 0) false
        pure (s, k+v)
      else if c == ``Add.add && nargs == 4 then do
        let v      ← evalNat (e.getArg! 3)
        let (s, k) ← getOffsetAux (e.getArg! 2) false
        pure (s, k+v)
      else if c == ``HAdd.hAdd && nargs == 6 then do
        let v      ← evalNat (e.getArg! 5)
        let (s, k) ← getOffsetAux (e.getArg! 4) false
        pure (s, k+v)
      else if top then failure else pure (e, 0)
    | _ => if top then failure else pure (e, 0)
  | e, top => if top then failure else pure (e, 0)

private def getOffset (e : Expr) : OptionT MetaM (Expr × Nat) :=
  getOffsetAux e true

private partial def isOffset : Expr → OptionT MetaM (Expr × Nat)
  | e@(Expr.app _ a _) =>
    let f := e.getAppFn
    match f with
    | Expr.mvar .. => withInstantiatedMVars e isOffset
    | Expr.const c _ _ =>
      let nargs := e.getAppNumArgs
      if (c == ``Nat.succ && nargs == 1) || (c == ``Nat.add && nargs == 2) || (c == ``Add.add && nargs == 4) || (c == ``HAdd.hAdd && nargs == 6) then
        getOffset e
      else
        failure
    | _ => failure
  | _ => failure

private def isNatZero (e : Expr) : MetaM Bool := do
  match (← evalNat e) with
```

```
  | some v => v == 0
  | _      => false

private def mkOffset (e : Expr) (offset : Nat) : MetaM Expr := do
  if offset == 0 then
    return e
  else if (← isNatZero e) then
    return mkNatLit offset
  else
    return mkAppB (mkConst ``Nat.add) e (mkNatLit offset)

def isDefEqOffset (s t : Expr) : MetaM LBool := do
  let ifNatExpr (x : MetaM LBool) : MetaM LBool := do
    let type ← inferType s
    -- Remark: we use `withNewMCtxDepth` to make sure we don't assing metavariables when performing the `isDefEq` test
    if (← withNewMCtxDepth <| Meta.isExprDefEqAux type (mkConst ``Nat)) then
      x
    else
      return LBool.undef
  let isDefEq (s t) : MetaM LBool :=
    ifNatExpr <| toLBoolM <| Meta.isExprDefEqAux s t
  match (← isOffset s) with
  | some (s, k₁) =>
    match (← isOffset t) with
    | some (t, k₂) => -- s+k₁ =?= t+k₂
      if k₁ == k₂ then
        isDefEq s t
      else if k₁ < k₂ then
        isDefEq s (← mkOffset t (k₂ - k₁))
      else
        isDefEq (← mkOffset s (k₁ - k₂)) t
    | none =>
      match (← evalNat t) with
      | some v₂ => -- s+k₁ =?= v₂
        if v₂ ≥ k₁ then
          isDefEq s (mkNatLit $ v₂ - k₁)
        else
          ifNatExpr <| return LBool.false
      | none =>
        return LBool.undef
  | none =>
    match (← evalNat s) with
    | some v₁ =>
      match (← isOffset t) with
      | some (t, k₂) => -- v₁ =?= t+k₂
        if v₁ ≥ k₂ then
          isDefEq (mkNatLit $ v₁ - k₂) t
```

```
      else
        ifNatExpr <| return LBool.false
    | none =>
      match (← evalNat t) with
      | some v₂ => ifNatExpr <| return (v₁ == v₂).toLBool -- v₁ =?= v₂
      | none    => return LBool.undef
  | none => return LBool.undef

end Lean.Meta
// ::::::::::::::
// PPGoal.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: Leonardo de Moura
-/
import Lean.Hygiene
import Lean.Meta.InferType

namespace Lean.Meta

def ppAuxDeclsDefault := false
builtin_initialize
  registerOption `pp.auxDecls { defValue := ppAuxDeclsDefault, group := "pp", descr := "display auxiliary declarations used to compile recursive functions" }
def getAuxDeclsOption (o : Options) : Bool :=
  o.get `pp.auxDecls ppAuxDeclsDefault

def ppInaccessibleNamesDefault := false
builtin_initialize
  registerOption `pp.inaccessibleNames { defValue := ppInaccessibleNamesDefault, group := "pp", descr := "display inaccessible declarations in the local context" }
def getInaccessibleNamesOption (o : Options) : Bool :=
  o.get `pp.inaccessibleNames ppInaccessibleNamesDefault

namespace ToHide

structure State where
  hiddenInaccessibleProp : NameSet := {} -- FVarIds of Propostions with inaccessible names but containing only visible names. We show only their types
  hiddenInaccessible     : NameSet := {} -- FVarIds with inaccessible names, but not in hiddenInaccessibleProp
  modified               : Bool := false

structure Context where
  goalTarget : Expr

```

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

/- Return true if `fvarId` is marked as an hidden inaccessible or inaccessible proposition -/
def isMarked (fvarId : FVarId) : M Bool := do
  let s ← get
  return s.hiddenInaccessible.contains fvarId || s.hiddenInaccessibleProp.contains fvarId

/- If `fvarId` isMarked, then unmark it. -/
def unmark (fvarId : FVarId) : M Unit := do
  modify fun s => {
    hiddenInaccessible     := s.hiddenInaccessible.erase fvarId
    hiddenInaccessibleProp := s.hiddenInaccessibleProp.erase fvarId
    modified               := true
  }

def moveToHiddeProp (fvarId : FVarId) : M Unit := do
  modify fun s => {
    hiddenInaccessible     := s.hiddenInaccessible.erase fvarId
    hiddenInaccessibleProp := s.hiddenInaccessibleProp.insert fvarId
    modified               := true
  }


/- Return true if the given local declaration has a "visible dependency", that is, it contains
   a free variable that is `hiddenInaccessible`

   Recall that hiddenInaccessibleProps are visible, only their names are hidden -/
def hasVisibleDep (localDecl : LocalDecl) : M Bool := do
  let s ← get
  return (← getMCtx).findLocalDeclDependsOn localDecl fun fvarId =>
    !s.hiddenInaccessible.contains fvarId

/- Return true if the given local declaration has a "nonvisible dependency", that is, it contains
   a free variable that is `hiddenInaccessible` or `hiddenInaccessibleProp` -/
def hasInaccessibleNameDep (localDecl : LocalDecl) : M Bool := do
  let s ← get
  return (← getMCtx).findLocalDeclDependsOn localDecl fun fvarId =>
    s.hiddenInaccessible.contains fvarId || s.hiddenInaccessibleProp.contains fvarId

/- If `e` is visible, then any inaccessible in `e` marked as hidden should be unmarked. -/
partial def visitVisibleExpr (e : Expr) : M Unit := do
  visit (← instantiateMVars e) |>.run
where
  visit (e : Expr) : MonadCacheT Expr Unit M Unit := do
    if e.hasFVar then
      checkCache e fun _ => do
        match e with
```

```
        | Expr.forallE _ d b _   => visit d; visit b
        | Expr.lam _ d b _       => visit d; visit b
        | Expr.letE _ t v b _    => visit t; visit v; visit b
        | Expr.app f a _         => visit f; visit a
        | Expr.mdata _ b _       => visit b
        | Expr.proj _ _ b _      => visit b
        | Expr.fvar fvarId _     => if (← isMarked fvarId) then unmark fvarId
        | _                      => pure ()

def fixpointStep : M Unit := do
  visitVisibleExpr (← read).goalTarget -- The goal target is a visible forward dependency
  (← getLCtx).forM fun localDecl => do
    let fvarId := localDecl.fvarId
    if (← get).hiddenInaccessible.contains fvarId then
      if (← hasVisibleDep localDecl) then
        /- localDecl is marked to be hidden, but it has a (backward) visible dependency. -/
        unmark fvarId
      if (← isProp localDecl.type) then
        unless (← hasInaccessibleNameDep localDecl) do
          moveToHiddeProp fvarId
    else
      visitVisibleExpr localDecl.type
      match localDecl.value? with
      | some value => visitVisibleExpr value
      | _ => pure ()

partial def fixpoint : M Unit := do
  modify fun s => { s with modified := false }
  fixpointStep
  if (← get).modified then
    fixpoint

/-
If pp.inaccessibleNames == false, then collect two sets of `FVarId`s : `hiddenInaccessible` and `hiddenInaccessibleProp`
1- `hiddenInaccessible` contains `FVarId`s of free variables with inaccessible names that
    a) are not propositions or are propositions containing "visible" names.
2- `hiddenInaccessibleProp` contains `FVarId`s of free variables with inaccessible names that are propositions
   containing "visible" names.
Both sets do not contain `FVarId`s that contain visible backward or forward dependencies.
The `goalTarget` counts as a forward dependency.

We say a name is visible if it is a free variable with FVarId not in `hiddenInaccessible` nor `hiddenInaccessibleProp`
-/
def collect (goalTarget : Expr) : MetaM (NameSet × NameSet) := do
  if getInaccessibleNamesOption (← getOptions) then
    /- Don't hide inaccessible names when `pp.inaccessibleNames` is set to true. -/
    return ({}, {})
```

```
  else
    let lctx ← getLCtx
    let hiddenInaccessible := lctx.foldl (init := {}) fun hiddenInaccessible localDecl => do
      if isInaccessibleUserName localDecl.userName then
        hiddenInaccessible.insert localDecl.fvarId
      else
        hiddenInaccessible
    let (_, s) ← fixpoint.run { goalTarget := goalTarget } |>.run { hiddenInaccessible := hiddenInaccessible }
    return (s.hiddenInaccessible, s.hiddenInaccessibleProp)

end ToHide

private def addLine (fmt : Format) : Format :=
  if fmt.isNil then fmt else fmt ++ Format.line

def ppGoal (mvarId : MVarId) : MetaM Format := do
  match (← getMCtx).findDecl? mvarId with
  | none          => pure "unknown goal"
  | some mvarDecl => do
    let indent         := 2 -- Use option
    let ppAuxDecls     := getAuxDeclsOption (← getOptions)
    let lctx           := mvarDecl.lctx
    let lctx           := lctx.sanitizeNames.run' { options := (← getOptions) }
    withLCtx lctx mvarDecl.localInstances do
      let (hidden, hiddenProp) ← ToHide.collect mvarDecl.type
      -- The followint two `let rec`s are being used to control the generated code size.
      -- Then should be remove after we rewrite the compiler in Lean
      let rec pushPending (ids : List Name) (type? : Option Expr) (fmt : Format) : MetaM Format :=
        if ids.isEmpty then
          pure fmt
        else
          let fmt := addLine fmt
          match ids, type? with
          | [], _        => pure fmt
          | _, none      => pure fmt
          | _, some type => do
            let typeFmt ← ppExpr type
            pure $ fmt ++ (Format.joinSep ids.reverse " " ++ " :" ++ Format.nest indent (Format.line ++ typeFmt)).group
      let rec ppVars (varNames : List Name) (prevType? : Option Expr) (fmt : Format) (localDecl : LocalDecl) : MetaM (List Name × Option Expr × Format) := do
        if hiddenProp.contains localDecl.fvarId then
          let fmt ← pushPending varNames prevType? fmt
          let fmt  := addLine fmt
          let type ← instantiateMVars localDecl.type
          let typeFmt ← ppExpr type
          let fmt  := fmt ++ " : " ++ typeFmt
          pure ([], none, fmt)
```

```
        else
          match localDecl with
          | LocalDecl.cdecl _ _ varName type _   =>
            let varName := varName.simpMacroScopes
            let type ← instantiateMVars type
            if prevType? == none || prevType? == some type then
              pure (varName :: varNames, some type, fmt)
            else do
              let fmt ← pushPending varNames prevType? fmt
              pure ([varName], some type, fmt)
          | LocalDecl.ldecl _ _ varName type val _ => do
            let varName := varName.simpMacroScopes
            let fmt ← pushPending varNames prevType? fmt
            let fmt  := addLine fmt
            let type ← instantiateMVars type
            let val  ← instantiateMVars val
            let typeFmt ← ppExpr type
            let valFmt ← ppExpr val
            let fmt  := fmt ++ (format varName ++ " : " ++ typeFmt ++ " :=" ++ Format.nest indent (Format.line ++ valFmt)).group
            pure ([], none, fmt)
      let (varNames, type?, fmt) ← lctx.foldlM (init := ([], none, Format.nil)) fun (varNames, prevType?, fmt) (localDecl : LocalDecl) =>
         if !ppAuxDecls && localDecl.isAuxDecl || hidden.contains localDecl.fvarId then
           pure (varNames, prevType?, fmt)
         else
           ppVars varNames prevType? fmt localDecl
      let fmt ← pushPending varNames type? fmt
      let fmt := addLine fmt
      let typeFmt ← ppExpr mvarDecl.type
      let fmt := fmt ++ "⊢" ++ " " ++ Format.nest indent typeFmt
      match mvarDecl.userName with
      | Name.anonymous => pure fmt
      | name           => return "case " ++ format name.eraseMacroScopes ++ Format.line ++ fmt

end Lean.Meta
// :::::::::::::
// RecursorInfo.lean
// :::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.AuxRecursor
import Lean.Util.FindExpr
import Lean.Meta.ExprDefEq

```

```
namespace Lean.Meta

inductive RecursorUnivLevelPos where
  | motive                -- marks where the universe of the motive should go
  | majorType (idx : Nat) -- marks where the #idx universe of the major premise type goes

instance : ToString RecursorUnivLevelPos := ⟨fun
  | RecursorUnivLevelPos.motive        => "<motive-univ>"
  | RecursorUnivLevelPos.majorType idx => toString idx⟩

structure RecursorInfo where
  recursorName  : Name
  typeName      : Name
  univLevelPos  : List RecursorUnivLevelPos
  depElim       : Bool
  recursive     : Bool
  numArgs       : Nat -- Total number of arguments
  majorPos      : Nat
  paramsPos     : List (Option Nat) -- Position of the recursor parameters in the major premise, instance implicit arguments are `none`
  indicesPos    : List Nat -- Position of the recursor indices in the major premise
  produceMotive : List Bool -- If the i-th element is true then i-th minor premise produces the motive

namespace RecursorInfo

def numParams (info : RecursorInfo) : Nat := info.paramsPos.length
def numIndices (info : RecursorInfo) : Nat := info.indicesPos.length
def motivePos (info : RecursorInfo) : Nat := info.numParams
def firstIndexPos (info : RecursorInfo) : Nat := info.majorPos - info.numIndices

def isMinor (info : RecursorInfo) (pos : Nat) : Bool :=
  if pos ≤ info.motivePos then false
  else if info.firstIndexPos ≤ pos && pos ≤ info.majorPos then false
  else true

def numMinors (info : RecursorInfo) : Nat :=
  let r := info.numArgs
  let r := r - info.motivePos - 1
  r - (info.majorPos + 1 - info.firstIndexPos)

instance : ToString RecursorInfo := ⟨fun info =>
  "{\n" ++
  "  name           := " ++ toString info.recursorName ++ "\n" ++
  "  type           := " ++ toString info.typeName ++ "\n" ++
  "  univs          := " ++ toString info.univLevelPos ++ "\n" ++
  "  depElim        := " ++ toString info.depElim ++ "\n" ++
  "  recursive      := " ++ toString info.recursive ++ "\n" ++
  "  numArgs        := " ++ toString info.numArgs ++ "\n" ++
```

```
  "  numParams      := " ++ toString info.numParams ++ "\n" ++
  "  numIndices     := " ++ toString info.numIndices ++ "\n" ++
  "  numMinors      := " ++ toString info.numMinors ++ "\n" ++
  "  major          := " ++ toString info.majorPos ++ "\n" ++
  "  motive         := " ++ toString info.motivePos ++ "\n" ++
  "  paramsAtMajor  := " ++ toString info.paramsPos ++ "\n" ++
  "  indicesAtMajor := " ++ toString info.indicesPos ++ "\n" ++
  "  produceMotive  := " ++ toString info.produceMotive ++ "\n" ++
  "}")

end RecursorInfo

private def mkRecursorInfoForKernelRec (declName : Name) (val : RecursorVal) : MetaM RecursorInfo := do
  let ival ← getConstInfoInduct val.getInduct
  let numLParams    := ival.levelParams.length
  let univLevelPos  := (List.range numLParams).map RecursorUnivLevelPos.majorType
  let univLevelPos  := if val.levelParams.length == numLParams then univLevelPos else RecursorUnivLevelPos.motive :: univLevelPos
  let produceMotive := List.replicate val.numMinors true
  let paramsPos     := (List.range val.numParams).map some
  let indicesPos    := (List.range val.numIndices).map fun pos => val.numParams + pos
  let numArgs       := val.numIndices + val.numParams + val.numMinors + val.numMotives + 1
  pure {
    recursorName  := declName,
    typeName      := val.getInduct,
    univLevelPos  := univLevelPos,
    majorPos      := val.getMajorIdx,
    depElim       := true,
    recursive     := ival.isRec,
    produceMotive := produceMotive,
    paramsPos     := paramsPos,
    indicesPos    := indicesPos,
    numArgs       := numArgs
  }


private def getMajorPosIfAuxRecursor? (declName : Name) (majorPos? : Option Nat) : MetaM (Option Nat) :=
  if majorPos?.isSome then pure majorPos?
  else do
    let env ← getEnv
    if !isAuxRecursor env declName then pure none
    else match declName with
    | Name.str p s _ =>
      if s != recOnSuffix && s != casesOnSuffix && s != brecOnSuffix then
        pure none
      else do
        let val ← getConstInfoRec (mkRecName p)
        pure $ some (val.numParams + val.numIndices + (if s == casesOnSuffix then 1 else val.numMotives))
```

```
  | _ => pure none

private def checkMotive (declName : Name) (motive : Expr) (motiveArgs : Array Expr) : MetaM Unit :=
  unless motive.isFVar && motiveArgs.all Expr.isFVar do
    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"

/- Compute number of parameters for (user-defined) recursor.
   We assume a parameter is anything that occurs before the motive -/
private partial def getNumParams (xs : Array Expr) (motive : Expr) (i : Nat) : Nat :=
  if h : i < xs.size then
    let x := xs.get ⟨i, h⟩
    if motive == x then i
    else getNumParams xs motive (i+1)
  else
    i

private def getMajorPosDepElim (declName : Name) (majorPos? : Option Nat) (xs : Array Expr) (motive : Expr) (motiveArgs : Array Expr)
    : MetaM (Expr × Nat × Bool) := do
  match majorPos? with
  | some majorPos =>
    if h : majorPos < xs.size then
      let major   := xs.get ⟨majorPos, h⟩
      let depElim := motiveArgs.contains major
      pure (major, majorPos, depElim)
    else throwError! "invalid major premise position for user defined recursor, recursor has only {xs.size} arguments"
  | none => do
    if motiveArgs.isEmpty then
      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)"
    let major := motiveArgs.back
    match xs.getIdx? major with
    | some majorPos => pure (major, majorPos, true)
    | none          => throwError! "ill-formed recursor '{declName}'"

private def getParamsPos (declName : Name) (xs : Array Expr) (numParams : Nat) (Iargs : Array Expr) : MetaM (List (Option Nat)) := do
  let mut paramsPos := #[]
  for i in [:numParams] do
    let x := xs[i]
    match (← Iargs.findIdxM? fun Iarg => isDefEq Iarg x) with
    | some j => paramsPos := paramsPos.push (some j)
    | none   => do
      let localDecl ← getLocalDecl x.fvarId!
      if localDecl.binderInfo.isInstImplicit then
        paramsPos := paramsPos.push none
      else
        throwError!"invalid user defined recursor '{declName}', type of the major premise does not contain the recursor parameter"
```

```
  pure paramsPos.toList

private def getIndicesPos (declName : Name) (xs : Array Expr) (majorPos numIndices : Nat) (Iargs : Array Expr) : MetaM (List Nat) := do
  let mut indicesPos := #[]
  for i in [:numIndices] do
    let i := majorPos - numIndices + i
    let x := xs[i]
    match (← Iargs.findIdxM? fun Iarg => isDefEq Iarg x) with
    | some j => indicesPos := indicesPos.push j
    | none   => throwError! "invalid user defined recursor '{declName}', type of the major premise does not contain the recursor index"
  pure indicesPos.toList

private def getMotiveLevel (declName : Name) (motiveResultType : Expr) : MetaM Level :=
  match motiveResultType with
  | Expr.sort u@(Level.zero _) _    => pure u
  | Expr.sort u@(Level.param _ _) _ => pure u
  | _                               =>
    throwError! "invalid user defined recursor '{declName}', motive result sort must be Prop or (Sort u) where u is a universe level parameter"

private def getUnivLevelPos (declName : Name) (lparams : List Name) (motiveLvl : Level) (Ilevels : List Level) : MetaM (List RecursorUnivLevelPos) := do
  let Ilevels := Ilevels.toArray
  let mut univLevelPos := #[]
  for p in lparams do
    if motiveLvl == mkLevelParam p then
      univLevelPos := univLevelPos.push RecursorUnivLevelPos.motive
    else
      match Ilevels.findIdx? fun u => u == mkLevelParam p with
      | some i => univLevelPos := univLevelPos.push (RecursorUnivLevelPos.majorType i)
      | none   =>
        throwError! "invalid user defined recursor '{declName}', major premise type does not contain universe level parameter '{p}'"
  pure univLevelPos.toList

private def getProduceMotiveAndRecursive (xs : Array Expr) (numParams numIndices majorPos : Nat) (motive : Expr) : MetaM (List Bool × Bool) := do
  let mut produceMotive := #[]
  let mut recursor      := false
  for i in [:xs.size] do
    if i < numParams + 1 then
      continue --skip parameters and motive
    if majorPos - numIndices ≤ i && i ≤ majorPos then
      continue -- skip indices and major premise
    -- process minor premise
    let x := xs[i]
    let xType ← inferType x
    (produceMotive, recursor) ← forallTelescopeReducing xType fun minorArgs minorResultType => minorResultType.withApp fun res _ => do
```

```
      let produceMotive := produceMotive.push (res == motive)
      let recursor ← if recursor then pure recursor else minorArgs.anyM fun minorArg => do
        let minorArgType ← inferType minorArg
        pure (minorArgType.find? fun e => e == motive).isSome
      pure (produceMotive, recursor)
  pure (produceMotive.toList, recursor)

private def checkMotiveResultType (declName : Name) (motiveArgs : Array Expr) (motiveResultType : Expr) (motiveTypeParams : Array Expr) : MetaM Unit := do
  if !motiveResultType.isSort || motiveArgs.size != motiveTypeParams.size then
    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"

private def mkRecursorInfoAux (cinfo : ConstantInfo) (majorPos? : Option Nat) : MetaM RecursorInfo := do
  let declName := cinfo.name
  let majorPos? ← getMajorPosIfAuxRecursor? declName majorPos?
  forallTelescopeReducing cinfo.type fun xs type => type.withApp fun motive motiveArgs => do
    checkMotive declName motive motiveArgs
    let numParams := getNumParams xs motive 0
    let (major, majorPos, depElim) ← getMajorPosDepElim declName majorPos? xs motive motiveArgs
    let numIndices := if depElim then motiveArgs.size - 1 else motiveArgs.size
    if majorPos < numIndices then
      throwError! "invalid user defined recursor '{declName}', indices must occur before major premise"
    let majorType ← inferType major
    majorType.withApp fun I Iargs =>
    match I with
    | Expr.const Iname Ilevels _ => do
      let paramsPos ← getParamsPos declName xs numParams Iargs
      let indicesPos ← getIndicesPos declName xs majorPos numIndices Iargs
      let motiveType ← inferType motive
      forallTelescopeReducing motiveType fun motiveTypeParams motiveResultType => do
        checkMotiveResultType declName motiveArgs motiveResultType motiveTypeParams
        let motiveLvl ← getMotiveLevel declName motiveResultType
        let univLevelPos ← getUnivLevelPos declName cinfo.levelParams motiveLvl Ilevels
        let (produceMotive, recursive) ← getProduceMotiveAndRecursive xs numParams numIndices majorPos motive
        pure {
          recursorName  := declName,
          typeName      := Iname,
          univLevelPos  := univLevelPos,
          majorPos      := majorPos,
          depElim       := depElim,
          recursive     := recursive,
          produceMotive := produceMotive,
          paramsPos     := paramsPos,
          indicesPos    := indicesPos,
          numArgs       := xs.size
```

```
        }
    | _ => throwError! "invalid user defined recursor '{declName}', type of the major premise must be of the form (I ...), where I is a constant"

/-
@[builtinAttrParser] def «recursor» := parser! "recursor " >> numLit
-/
def Attribute.Recursor.getMajorPos (stx : Syntax) : AttrM Nat := do
  if stx.getKind == `Lean.Parser.Attr.recursor then
    let pos := stx[1].isNatLit?.getD 0
    if pos == 0 then
      throwErrorAt! stx "major premise position must be greater than zero"
    return pos - 1
  else
    throwErrorAt! stx "unexpected attribute argument, numeral expected"

private def mkRecursorInfoCore (declName : Name) (majorPos? : Option Nat := none) : MetaM RecursorInfo := do
  let cinfo ← getConstInfo declName
  match cinfo with
  | ConstantInfo.recInfo val => mkRecursorInfoForKernelRec declName val
  | _                        => mkRecursorInfoAux cinfo majorPos?

builtin_initialize recursorAttribute : ParametricAttribute Nat ←
  registerParametricAttribute {
    name := `recursor,
    descr := "user defined recursor, numerical parameter specifies position of the major premise",
    getParam := fun _ stx => Attribute.Recursor.getMajorPos stx
    afterSet := fun declName majorPos => do
      discard <| mkRecursorInfoCore declName (some majorPos) |>.run'
  }

def getMajorPos? (env : Environment) (declName : Name) : Option Nat :=
  recursorAttribute.getParam env declName

def mkRecursorInfo (declName : Name) (majorPos? : Option Nat := none) : MetaM RecursorInfo := do
  let cinfo ← getConstInfo declName
  match cinfo with
  | ConstantInfo.recInfo val => mkRecursorInfoForKernelRec declName val
  | _                        => match majorPos? with
    | none => do mkRecursorInfoAux cinfo (getMajorPos? (← getEnv) declName)
    | _    => mkRecursorInfoAux cinfo majorPos?

end Lean.Meta
// ::::::::::::::
// ReduceEval.lean
// ::::::::::::::
/-
```

```
Copyright (c) 2020 Sebastian Ullrich. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Sebastian Ullrich
-/

/-! Evaluation by reduction -/

import Lean.Meta.Offset

namespace Lean.Meta

class ReduceEval (α : Type) where
  reduceEval : Expr → MetaM α

def reduceEval [ReduceEval α] (e : Expr) : MetaM α :=
  withAtLeastTransparency TransparencyMode.default $
  ReduceEval.reduceEval e

private def throwFailedToEval (e : Expr) : MetaM α :=
  throwError! "reduceEval: failed to evaluate argument{indentExpr e}"

instance : ReduceEval Nat where
  reduceEval e := do
    let e ← whnf e
    let some n ← evalNat e | throwFailedToEval e
    pure n

instance [ReduceEval α] : ReduceEval (Option α) where
  reduceEval e := do
    let e ← whnf e
    let Expr.const c .. ← pure e.getAppFn | throwFailedToEval e
    let nargs := e.getAppNumArgs
    if      c == `Option.none && nargs == 0 then pure none
    else if c == `Option.some && nargs == 1 then some <$> reduceEval e.appArg!
    else throwFailedToEval e

instance : ReduceEval String where
  reduceEval e := do
    let Expr.lit (Literal.strVal s) _ ← whnf e | throwFailedToEval e
    pure s

private partial def evalName (e : Expr) : MetaM Name := do
  let e ← whnf e
  let Expr.const c _ _ ← pure e.getAppFn | throwFailedToEval e
  let nargs := e.getAppNumArgs
  if      c == `Lean.Name.anonymous && nargs == 0 then pure Name.anonymous
  else if c == `Lean.Name.str && nargs == 3 then do
```

```
    let n ← evalName $ e.getArg! 0
    let s ← reduceEval $ e.getArg! 1
    pure $ Name.mkStr n s
  else if c == `Lean.Name.num && nargs == 3 then do
    let n ← evalName $ e.getArg! 0
    let u ← reduceEval $ e.getArg! 1
    pure $ Name.mkNum n u
  else
    throwFailedToEval e

instance : ReduceEval Name where
  reduceEval := evalName

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

namespace Lean.Meta

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

```
                args ← args.modifyM i visit
            else
              args ← args.modifyM i visit
          pure (mkAppN f args)
        | Expr.lam ..     => lambdaTelescope e fun xs b => do mkLambdaFVars xs (← visit b)
        | Expr.forallE .. => forallTelescope e fun xs b => do mkForallFVars xs (← visit b)
        | _               => return e
  visit e |>.run

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

namespace Lean.Meta

/-- Create `SizeOf` local instances for applicable parameters, and execute `k` using them. -/
private partial def mkLocalInstances {α} (params : Array Expr) (k : Array Expr → MetaM α) : MetaM α :=
  loop 0 #[]
where
  loop (i : Nat) (insts : Array Expr) : MetaM α := do
    if i < params.size then
      let param := params[i]
      let paramType ← inferType param
      let instType? ← forallTelescopeReducing paramType fun xs _ => do
        let type ← mkAppN param xs
        try
          let sizeOf ← mkAppM `SizeOf #[type]
          let instType ← mkForallFVars xs sizeOf
          return some instType
        catch _ =>
          return none
      match instType? with
      | none => loop (i+1) insts
      | some instType =>
        let instName ← mkFreshUserName `inst
        withLocalDecl instName BinderInfo.instImplicit instType fun inst =>
          loop (i+1) (insts.push inst)
    else
      k insts
```

```

/--
  Return `some x` if `fvar` has type of the form `... -> motive ... fvar` where `motive` in `motiveFVars`.
  That is, `x` "produces" one of the recursor motives.
-/
private def isInductiveHypothesis? (motiveFVars : Array Expr) (fvar : Expr) : MetaM (Option Expr) := do
  forallTelescopeReducing (← inferType fvar) fun _ type =>
    if type.isApp && motiveFVars.contains type.getAppFn then
      return some type.appArg!
    else
      return none

private def isInductiveHypothesis (motiveFVars : Array Expr) (fvar : Expr) : MetaM Bool :=
  return (← isInductiveHypothesis? motiveFVars fvar).isSome

/--
  Let `motiveFVars` be free variables for each motive in a kernel recursor, and `minorFVars` the free variables for a minor premise.
  Then, return `some idx` if `minorFVars[idx]` has a type of the form `... -> motive ... fvar` for some `motive` in `motiveFVars`.
-/
private def isRecField? (motiveFVars : Array Expr) (minorFVars : Array Expr) (fvar : Expr) : MetaM (Option Nat) := do
  let mut idx := 0
  for minorFVar in minorFVars do
    if let some fvar' ← isInductiveHypothesis? motiveFVars minorFVar then
      if fvar == fvar' then
        return some idx
    idx := idx + 1
  return none

private partial def mkSizeOfMotives {α} (motiveFVars : Array Expr) (k : Array Expr → MetaM α) : MetaM α :=
  loop 0 #[]
where
  loop (i : Nat) (motives : Array Expr) : MetaM α := do
    if i < motiveFVars.size then
      let type ← inferType motiveFVars[i]
      let motive ← forallTelescopeReducing type fun xs _ => do
        mkLambdaFVars xs <| mkConst ``Nat
      trace[Meta.sizeOf]! "motive: {motive}"
      loop (i+1) (motives.push motive)
    else
      k motives

private partial def mkSizeOfMinors {α} (motiveFVars : Array Expr) (minorFVars : Array Expr) (minorFVars' : Array Expr) (k : Array Expr → MetaM α) : MetaM α :=
  assert! minorFVars.size == minorFVars'.size
  loop 0 #[]
where
  loop (i : Nat) (minors : Array Expr) : MetaM α := do
```

```
    if i < minorFVars.size then
      forallTelescopeReducing (← inferType minorFVars[i]) fun xs _ =>
      forallBoundedTelescope (← inferType minorFVars'[i]) xs.size fun xs' _ => do
        let mut minor ← mkNumeral (mkConst ``Nat) 1
        for x in xs, x' in xs' do
          unless (← isInductiveHypothesis motiveFVars x) do
          unless (← whnf (← inferType x)).isForall do -- we suppress higher-order fields
            match (← isRecField? motiveFVars xs x) with
            | some idx => minor ← mkAdd minor xs'[idx]
            | none     => minor ← mkAdd minor (← mkAppM ``SizeOf.sizeOf #[x'])
        minor ← mkLambdaFVars xs' minor
        trace[Meta.sizeOf]! "minor: {minor}"
        loop (i+1) (minors.push minor)
    else
      k minors

/--
  Create a "sizeOf" function with name `declName` using the recursor `recName`.
-/
partial def mkSizeOfFn (recName : Name) (declName : Name): MetaM Unit := do
  trace[Meta.sizeOf]! "recName: {recName}"
  let recInfo : RecursorVal ← getConstInfoRec recName
  forallTelescopeReducing recInfo.type fun xs type =>
    let levelParams := recInfo.levelParams.tail! -- universe parameters for declaration being defined
    let params := xs[:recInfo.numParams]
    let motiveFVars := xs[recInfo.numParams : recInfo.numParams + recInfo.numMotives]
    let minorFVars := xs[recInfo.getFirstMinorIdx : recInfo.getFirstMinorIdx + recInfo.numMinors]
    let indices := xs[recInfo.getFirstIndexIdx : recInfo.getFirstIndexIdx + recInfo.numIndices]
    let major := xs[recInfo.getMajorIdx]
    let nat := mkConst ``Nat
    mkLocalInstances params fun localInsts =>
    mkSizeOfMotives motiveFVars fun motives => do
      let us := levelOne :: levelParams.map mkLevelParam -- universe level parameters for `rec`-application
      let recFn := mkConst recName us
      let val := mkAppN recFn (params ++ motives)
      forallBoundedTelescope (← inferType val) recInfo.numMinors fun minorFVars' _ =>
      mkSizeOfMinors motiveFVars minorFVars minorFVars' fun minors => do
        let sizeOfParams := params ++ localInsts ++ indices ++ #[major]
        let sizeOfType ← mkForallFVars sizeOfParams nat
        let val := mkAppN val (minors ++ indices ++ #[major])
        trace[Meta.sizeOf]! "val: {val}"
        let sizeOfValue ← mkLambdaFVars sizeOfParams val
        addDecl <| Declaration.defnDecl {
          name        := declName
          levelParams := levelParams
          type        := sizeOfType
          value       := sizeOfValue
```

```
          safety      := DefinitionSafety.safe
          hints       := ReducibilityHints.abbrev
        }

/--
  Create `sizeOf` functions for all inductive datatypes in the mutual inductive declaration containing `typeName`
  The resulting array contains the generated functions names. The `NameMap` maps recursor names into the generated function names.
  There is a function for each element of the mutual inductive declaration, and for auxiliary recursors for nested inductive types.
-/
def mkSizeOfFns (typeName : Name) : MetaM (Array Name × NameMap Name) := do
  let indInfo ← getConstInfoInduct typeName
  let recInfo ← getConstInfoRec (mkRecName typeName)
  let numExtra := recInfo.numMotives - indInfo.all.length -- numExtra > 0 for nested inductive types
  let mut result := #[]
  let baseName := indInfo.all.head! ++ `_sizeOf -- we use the first inductive type as the base name for `sizeOf` functions
  let mut i := 1
  let mut recMap : NameMap Name := {}
  for indTypeName in indInfo.all do
    let sizeOfName := baseName.appendIndexAfter i
    let recName := mkRecName indTypeName
    mkSizeOfFn recName sizeOfName
    recMap := recMap.insert recName sizeOfName
    result := result.push sizeOfName
    i := i + 1
  for j in [:numExtra] do
    let recName := (mkRecName indInfo.all.head!).appendIndexAfter (j+1)
    let sizeOfName := baseName.appendIndexAfter i
    mkSizeOfFn recName sizeOfName
    recMap := recMap.insert recName sizeOfName
    result := result.push sizeOfName
    i := i + 1
  return (result, recMap)

def mkSizeOfSpecLemmaName (ctorName : Name) : Name :=
  ctorName ++ `sizeOf_spec

def mkSizeOfSpecLemmaInstance (ctorApp : Expr) : MetaM Expr :=
  matchConstCtor ctorApp.getAppFn (fun _ => throwError! "failed to apply 'sizeOf' spec, constructor expected{indentExpr ctorApp}") fun ctorInfo ctorLevels => do
    let ctorArgs     := ctorApp.getAppArgs
    let ctorFields   := ctorArgs[ctorArgs.size - ctorInfo.numFields:]
    let lemmaName  := mkSizeOfSpecLemmaName ctorInfo.name
    let lemmaInfo  ← getConstInfo lemmaName
    let lemmaArity ← forallTelescopeReducing lemmaInfo.type fun xs _ => return xs.size
    let lemmaArgMask := mkArray (lemmaArity - ctorInfo.numFields) (none (α := Expr))
    let lemmaArgMask := lemmaArgMask ++ ctorFields.toArray.map some
    mkAppOptM lemmaName lemmaArgMask
```

```

/- SizeOf spec theorem for nested inductive types -/
namespace SizeOfSpecNested

structure Context where
  indInfo    : InductiveVal
  sizeOfFns  : Array Name
  ctorName   : Name
  params     : Array Expr
  localInsts : Array Expr
  recMap     : NameMap Name -- mapping from recursor name into `_sizeOf_<idx>` function name (see `mkSizeOfFns`)

abbrev M := ReaderT Context MetaM

def throwUnexpected {α} (msg : MessageData) : M α := do
  throwError! "failed to generate sizeOf lemma for {(← read).ctorName} (use `set_option genSizeOfSpec false` to disable lemma generation), {msg}"

def throwFailed {α} : M α := do
  throwError! "failed to generate sizeOf lemma for {(← read).ctorName}, (use `set_option genSizeOfSpec false` to disable lemma generation)"

/-- Convert a recursor application into a `_sizeOf_<idx>` application. -/
private def recToSizeOf (e : Expr) : M Expr := do
  matchConstRec e.getAppFn (fun _ => throwFailed) fun info us => do
    match (← read).recMap.find? info.name with
    | none => throwUnexpected m!"expected recursor application {indentExpr e}"
    | some sizeOfName =>
      let args    := e.getAppArgs
      let indices := args[info.getFirstIndexIdx : info.getFirstIndexIdx + info.numIndices]
      let major   := args[info.getMajorIdx]
      return mkAppN (mkConst sizeOfName us.tail!) ((← read).params ++ (← read).localInsts ++ indices ++ #[major])

mutual
  /-- Construct minor premise proof for `mkSizeOfAuxLemmaProof`. `ys` contains fields and inductive hypotheses for the minor premise. -/
  private partial def mkMinorProof (ys : Array Expr) (lhs rhs : Expr) : M Expr := do
    trace[Meta.sizeOf.minor]! "{lhs} =?= {rhs}"
    if (← isDefEq lhs rhs) then
      mkEqRefl rhs
    else
      match (← whnfI lhs).natAdd?, (← whnfI rhs).natAdd? with
      | some (a₁, b₁), some (a₂, b₂) =>
        let p₁ ← mkMinorProof ys a₁ a₂
        let p₂ ← mkMinorProofStep ys b₁ b₂
        mkCongr (← mkCongrArg (mkConst ``Nat.add) p₁) p₂
      | _, _ =>
```

```
      throwUnexpected m!"expected 'Nat.add' application, lhs is {indentExpr lhs}\nrhs is{indentExpr rhs}"

/--
  Helper method for `mkMinorProof`. The proof step is one of the following
  - Reflexivity
  - Assumption (i.e., using an inductive hypotheses from `ys`)
  - `mkSizeOfAuxLemma` application. This case happens when we have multiple levels of nesting
-/
private partial def mkMinorProofStep (ys : Array Expr) (lhs rhs : Expr) : M Expr := do
  if (← isDefEq lhs rhs) then
    mkEqRefl rhs
  else
    let lhs ← recToSizeOf lhs
    trace[Meta.sizeOf.minor.step]! "{lhs} =?= {rhs}"
    let target ← mkEq lhs rhs
    for y in ys do
      if (← isDefEq (← inferType y) target) then
        return y
    mkSizeOfAuxLemma lhs rhs

/-- Construct proof of auxiliary lemma. See `mkSizeOfAuxLemma` -/
private partial def mkSizeOfAuxLemmaProof (info : InductiveVal) (lhs rhs : Expr) : M Expr := do
  let lhsArgs := lhs.getAppArgs
  let sizeOfBaseArgs := lhsArgs[:lhsArgs.size - info.numIndices - 1]
  let indicesMajor := lhsArgs[lhsArgs.size - info.numIndices - 1:]
  let sizeOfLevels := lhs.getAppFn.constLevels!
  /- Auxiliary function for constructing an `_sizeOf_<idx>` for `ys`,
     where `ys` are the indices + major.
     Recall that if `info.name` is part of a mutually inductive declaration, then the resulting application
     is not necessarily a `lhs.getAppFn` application.
     The result is an application of one of the `(← read),sizeOfFns` functions.
     We use this auxiliary function to builtin the motive of the recursor. -/
  let rec mkSizeOf (ys : Array Expr) : M Expr := do
    for sizeOfFn in (← read).sizeOfFns do
      let candidate := mkAppN (mkAppN (mkConst sizeOfFn sizeOfLevels) sizeOfBaseArgs) ys
      if (← isTypeCorrect candidate) then
        return candidate
    throwFailed
  let major := lhs.appArg!
  let majorType ← whnf (← inferType major)
  let majorTypeArgs := majorType.getAppArgs
  match majorType.getAppFn.const? with
  | none => throwFailed
  | some (_, us) =>
    let recName := mkRecName info.name
    let recInfo ← getConstInfoRec recName
    let r := mkConst recName (levelZero :: us)
```

```
    let r := mkAppN r majorTypeArgs[:info.numParams]
    forallBoundedTelescope (← inferType r) recInfo.numMotives fun motiveFVars _ => do
      let mut r := r
      -- Add motives
      for motiveFVar in motiveFVars do
        let motive ← forallTelescopeReducing (← inferType motiveFVar) fun ys _ => do
          let lhs ← mkSizeOf ys
          let rhs ← mkAppM ``SizeOf.sizeOf #[ys.back]
          mkLambdaFVars ys (← mkEq lhs rhs)
        r := mkApp r motive
      forallBoundedTelescope (← inferType r) recInfo.numMinors fun minorFVars _ => do
        let mut r := r
        -- Add minors
        for minorFVar in minorFVars do
          let minor ← forallTelescopeReducing (← inferType minorFVar) fun ys target => do
            let target ← whnf target
            match target.eq? with
            | none => throwFailed
            | some (_, lhs, rhs) =>
              if (← isDefEq lhs rhs) then
                mkLambdaFVars ys (← mkEqRefl rhs)
              else
                let lhs ← unfoldDefinition lhs -- Unfold `_sizeOf_<idx>`
                -- rhs is of the form `sizeOf (ctor ...)`
                let ctorApp := rhs.appArg!
                let specLemma ← mkSizeOfSpecLemmaInstance ctorApp
                let specEq ← whnf (← inferType specLemma)
                match specEq.eq? with
                | none => throwFailed
                | some (_, rhs, rhsExpanded) =>
                  let lhs_eq_rhsExpanded ← mkMinorProof ys lhs rhsExpanded
                  let rhsExpanded_eq_rhs ← mkEqSymm specLemma
                  mkLambdaFVars ys (← mkEqTrans lhs_eq_rhsExpanded rhsExpanded_eq_rhs)
          r := mkApp r minor
        -- Add indices and major
        return mkAppN r indicesMajor

/--
  Generate proof for `C._sizeOf_<idx> t = sizeOf t` where `C._sizeOf_<idx>` is a auxiliary function
  generated for a nested inductive type in `C`.
  For example, given
  ```lean
  inductive Expr where
    | app (f : String) (args : List Expr)
  ```
  We generate the auxiliary function `Expr._sizeOf_1 : List Expr → Nat`.
  To generate the `sizeOf` spec lemma
```

````
  ```
  sizeOf (Expr.app f args) = 1 + sizeOf f + sizeOf args
  ```
  we need an auxiliary lemma for showing `Expr._sizeOf_1 args = sizeOf args`.
  Recall that `sizeOf (Expr.app f args)` is definitionally equal to `1 + sizeOf f + Expr._sizeOf_1 args`, but
  `Expr._sizeOf_1 args` is **not** definitionally equal to `sizeOf args`. We need a proof by induction.
-/
private partial def mkSizeOfAuxLemma (lhs rhs : Expr) : M Expr := do
  trace[Meta.sizeOf.aux]! "{lhs} =?= {rhs}"
  match lhs.getAppFn.const? with
  | none => throwFailed
  | some (fName, us) =>
    let thmLevelParams ← us.mapM fun
      | Level.param n _ => return n
      | _ => throwFailed
    let thmName  := fName.appendAfter "_eq"
    if (← getEnv).contains thmName then
      -- Auxiliary lemma has already been defined
      return mkAppN (mkConst thmName us) lhs.getAppArgs
    else
      -- Define auxiliary lemma
      -- First, generalize indices
      let x := lhs.appArg!
      let xType ← whnf (← inferType x)
      matchConstInduct xType.getAppFn (fun _ => throwFailed) fun info _ => do
        let params := xType.getAppArgs[:info.numParams]
        forallTelescopeReducing (← inferType (mkAppN xType.getAppFn params)) fun indices _ => do
          let majorType := mkAppN (mkAppN xType.getAppFn params) indices
          withLocalDeclD `x majorType fun major => do
            let lhsArgs := lhs.getAppArgs
            let lhsArgsNew := lhsArgs[:lhsArgs.size - 1 - indices.size] ++ indices ++ #[major]
            let lhsNew := mkAppN lhs.getAppFn lhsArgsNew
            let rhsNew ← mkAppM ``SizeOf.sizeOf #[major]
            let eq ← mkEq lhsNew rhsNew
            let thmParams := lhsArgsNew
            let thmType ← mkForallFVars thmParams eq
            let thmValue ← mkSizeOfAuxLemmaProof info lhsNew rhsNew
            let thmValue ← mkLambdaFVars thmParams thmValue
            trace[Meta.sizeOf]! "thmValue: {thmValue}"
            addDecl <| Declaration.thmDecl {
              name        := thmName
              levelParams := thmLevelParams
              type        := thmType
              value       := thmValue
            }
            return mkAppN (mkConst thmName us) lhs.getAppArgs
````

```
end

/- Prove SizeOf spec lemma of the form `sizeOf <ctor-application> = 1 + sizeOf <field_1> + ... + sizeOf <field_n> -/
partial def main (lhs rhs : Expr) : M Expr := do
  if (← isDefEq lhs rhs) then
    mkEqRefl rhs
  else
    /- Expand lhs and rhs to obtain `Nat.add` applications -/
    let lhs ← whnfI lhs            -- Expand `sizeOf (ctor ...)` into `_sizeOf_<idx>` application
    let lhs ← unfoldDefinition lhs -- Unfold `_sizeOf_<idx>` application into `HAdd.hAdd` application
    loop lhs rhs
where
  loop (lhs rhs : Expr) : M Expr := do
    trace[Meta.sizeOf.loop]! "{lhs} =?= {rhs}"
    if (← isDefEq lhs rhs) then
      mkEqRefl rhs
    else
      match (← whnfI lhs).natAdd?, (← whnfI rhs).natAdd? with
      | some (a₁, b₁), some (a₂, b₂) =>
        let p₁ ← loop a₁ a₂
        let p₂ ← step b₁ b₂
        mkCongr (← mkCongrArg (mkConst ``Nat.add) p₁) p₂
      | _, _ =>
        throwUnexpected m!"expected 'Nat.add' application, lhs is {indentExpr lhs}\nrhs is{indentExpr rhs}"

  step (lhs rhs : Expr) : M Expr := do
    if (← isDefEq lhs rhs) then
      mkEqRefl rhs
    else
      let lhs ← recToSizeOf lhs
      mkSizeOfAuxLemma lhs rhs

end SizeOfSpecNested

private def mkSizeOfSpecTheorem (indInfo : InductiveVal) (sizeOfFns : Array Name) (recMap : NameMap Name) (ctorName : Name) : MetaM Unit := do
  let ctorInfo ← getConstInfoCtor ctorName
  let us := ctorInfo.levelParams.map mkLevelParam
  forallTelescopeReducing ctorInfo.type fun xs _ => do
    let params := xs[:ctorInfo.numParams]
    let fields := xs[ctorInfo.numParams:]
    let ctorApp := mkAppN (mkConst ctorName us) xs
    mkLocalInstances params fun localInsts => do
      let lhs ← mkAppM ``SizeOf.sizeOf #[ctorApp]
      let mut rhs ← mkNumeral (mkConst ``Nat) 1
      for field in fields do
        unless (← whnf (← inferType field)).isForall do
```

```
          rhs ← mkAdd rhs (← mkAppM ``SizeOf.sizeOf #[field])
      let target ← mkEq lhs rhs
      let thmName   := mkSizeOfSpecLemmaName ctorName
      let thmParams := params ++ localInsts ++ fields
      let thmType ← mkForallFVars thmParams target
      let thmValue ←
        if indInfo.isNested then
          SizeOfSpecNested.main lhs rhs |>.run {
            indInfo := indInfo, sizeOfFns := sizeOfFns, ctorName := ctorName, params := params, localInsts := localInsts, recMap := recMap
          }
        else
          mkEqRefl rhs
      let thmValue ← mkLambdaFVars thmParams thmValue
      addDecl <| Declaration.thmDecl {
        name        := thmName
        levelParams := ctorInfo.levelParams
        type        := thmType
        value       := thmValue
      }

private def mkSizeOfSpecTheorems (indTypeNames : Array Name) (sizeOfFns : Array Name) (recMap : NameMap Name) : MetaM Unit := do
  for indTypeName in indTypeNames do
    let indInfo ← getConstInfoInduct indTypeName
    for ctorName in indInfo.ctors do
      mkSizeOfSpecTheorem indInfo sizeOfFns recMap ctorName
  return ()

register_builtin_option genSizeOf : Bool := {
  defValue := true
  descr    := "generate `SizeOf` instance for inductive types and structures"
}

register_builtin_option genSizeOfSpec : Bool := {
  defValue := true
  descr    := "generate `SizeOf` specificiation theorems for automatically generated instances"
}

def mkSizeOfInstances (typeName : Name) : MetaM Unit := do
  if (← getEnv).contains ``SizeOf && genSizeOf.get (← getOptions) && !(← isInductivePredicate typeName) then
    let indInfo ← getConstInfoInduct typeName
    unless indInfo.isUnsafe do
      let (fns, recMap) ← mkSizeOfFns typeName
      for indTypeName in indInfo.all, fn in fns do
        let indInfo ← getConstInfoInduct indTypeName
        forallTelescopeReducing indInfo.type fun xs _ =>
          let params := xs[:indInfo.numParams]
```

```
          let indices := xs[indInfo.numParams:]
          mkLocalInstances params fun localInsts => do
            let us := indInfo.levelParams.map mkLevelParam
            let indType := mkAppN (mkConst indTypeName us) xs
            let sizeOfIndType ← mkAppM ``SizeOf #[indType]
            withLocalDeclD `m indType fun m => do
              let v ← mkLambdaFVars #[m] <| mkAppN (mkConst fn us) (params ++ localInsts ++ indices ++ #[m])
              let sizeOfMk ← mkAppM ``SizeOf.mk #[v]
              let instDeclName := indTypeName ++ `_sizeOf_inst
              let instDeclType ← mkForallFVars (xs ++ localInsts) sizeOfIndType
              let instDeclValue ← mkLambdaFVars (xs ++ localInsts) sizeOfMk
              addDecl <| Declaration.defnDecl {
                name        := instDeclName
                levelParams := indInfo.levelParams
                type        := instDeclType
                value       := instDeclValue
                safety      := DefinitionSafety.safe
                hints       := ReducibilityHints.abbrev
              }
              addInstance instDeclName AttributeKind.global (evalPrio! default)
      if genSizeOfSpec.get (← getOptions) then
        mkSizeOfSpecTheorems indInfo.all.toArray fns recMap

builtin_initialize
  registerTraceClass `Meta.sizeOf

end Lean.Meta
// ::::::::::::::
// SynthInstance.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Daniel Selsam, Leonardo de Moura

Type class instance synthesizer using tabled resolution.
-/
import Lean.Meta.Basic
import Lean.Meta.Instances
import Lean.Meta.LevelDefEq
import Lean.Meta.AbstractMVars
import Lean.Meta.WHNF
import Lean.Util.Profile

namespace Lean.Meta

register_builtin_option synthInstance.maxHeartbeats : Nat := {
```

```
  defValue := 500
  descr := "maximum amount of heartbeats per typeclass resolution problem. A heartbeat is number of (small) memory allocations (in thousands), 0 means no limit"
}

register_builtin_option synthInstance.maxSize : Nat := {
  defValue := 128
  descr := "maximum number of instances used to construct a solution in the type class instance synthesis procedure"
}

namespace SynthInstance

def getMaxHeartbeats (opts : Options) : Nat :=
  synthInstance.maxHeartbeats.get opts * 1000

open Std (HashMap)

builtin_initialize inferTCGoalsRLAttr : TagAttribute ←
  registerTagAttribute `inferTCGoalsRL "instruct type class resolution procedure to solve goals from right to left for this instance"

def hasInferTCGoalsRLAttribute (env : Environment) (constName : Name) : Bool :=
  inferTCGoalsRLAttr.hasTag env constName

structure GeneratorNode where
  mvar            : Expr
  key             : Expr
  mctx            : MetavarContext
  instances       : Array Expr
  currInstanceIdx : Nat
  deriving Inhabited

structure ConsumerNode where
  mvar     : Expr
  key      : Expr
  mctx     : MetavarContext
  subgoals : List Expr
  size     : Nat -- instance size so far
  deriving Inhabited

inductive Waiter where
  | consumerNode : ConsumerNode → Waiter
  | root         : Waiter

def Waiter.isRoot : Waiter → Bool
  | Waiter.consumerNode _ => false
  | Waiter.root           => true

```

```
/-
  In tabled resolution, we creating a mapping from goals (e.g., `Coe Nat ?x`) to
  answers and waiters. Waiters are consumer nodes that are waiting for answers for a
  particular node.

  We implement this mapping using a `HashMap` where the keys are
  normalized expressions. That is, we replace assignable metavariables
  with auxiliary free variables of the form `_tc.<idx>`. We do
  not declare these free variables in any local context, and we should
  view them as "normalized names" for metavariables. For example, the
  term `f ?m ?m ?n` is normalized as
  `f _tc.0 _tc.0 _tc.1`.

  This approach is structural, and we may visit the same goal more
  than once if the different occurrences are just definitionally
  equal, but not structurally equal.

  Remark: a metavariable is assignable only if its depth is equal to
  the metavar context depth.
-/
namespace  MkTableKey

structure State where
  nextIdx : Nat := 0
  lmap    : HashMap MVarId Level := {}
  emap    : HashMap MVarId Expr := {}

abbrev M := ReaderT MetavarContext (StateM State)

partial def normLevel (u : Level) : M Level := do
  if !u.hasMVar then
    pure u
  else match u with
    | Level.succ v _      => return u.updateSucc! (← normLevel v)
    | Level.max v w _     => return u.updateMax! (← normLevel v) (← normLevel w)
    | Level.imax v w _    => return u.updateIMax! (← normLevel v) (← normLevel w)
    | Level.mvar mvarId _ =>
      let mctx ← read
      if !mctx.isLevelAssignable mvarId then
        pure u
      else
        let s ← get
        match s.lmap.find? mvarId with
        | some u' => pure u'
        | none    =>
          let u' := mkLevelParam $ Name.mkNum `_tc s.nextIdx
          modify fun s => { s with nextIdx := s.nextIdx + 1, lmap := s.lmap.insert mvarId u' }
```

```
        pure u'
    | u => pure u

partial def normExpr (e : Expr) : M Expr := do
  if !e.hasMVar then
    pure e
  else match e with
    | Expr.const _ us _    => return e.updateConst! (← us.mapM normLevel)
    | Expr.sort u _        => return e.updateSort! (← normLevel u)
    | Expr.app f a _       => return e.updateApp! (← normExpr f) (← normExpr a)
    | Expr.letE _ t v b _  => return e.updateLet! (← normExpr t) (← normExpr v) (← normExpr b)
    | Expr.forallE _ d b _ => return e.updateForallE! (← normExpr d) (← normExpr b)
    | Expr.lam _ d b _     => return e.updateLambdaE! (← normExpr d) (← normExpr b)
    | Expr.mdata _ b _     => return e.updateMData! (← normExpr b)
    | Expr.proj _ _ b _    => return e.updateProj! (← normExpr b)
    | Expr.mvar mvarId _   =>
      let mctx ← read
      if !mctx.isExprAssignable mvarId then
        pure e
      else
        let s ← get
        match s.emap.find? mvarId with
        | some e' => pure e'
        | none    => do
          let e' := mkFVar $ Name.mkNum `_tc s.nextIdx
          modify fun s => { s with nextIdx := s.nextIdx + 1, emap := s.emap.insert mvarId e' }
          pure e'
    | _ => pure e

end MkTableKey

/- Remark: `mkTableKey` assumes `e` does not contain assigned metavariables. -/
def mkTableKey (mctx : MetavarContext) (e : Expr) : Expr :=
  MkTableKey.normExpr e mctx |>.run' {}

structure Answer where
  result     : AbstractMVarsResult
  resultType : Expr
  size       : Nat

instance : Inhabited Answer where
  default := { result := arbitrary, resultType := arbitrary, size := 0 }

structure TableEntry where
  waiters : Array Waiter
  answers : Array Answer := #[]

```

```
structure Context where
  maxResultSize : Nat
  maxHeartbeats : Nat

/-
  Remark: the SynthInstance.State is not really an extension of `Meta.State`.
  The field `postponed` is not needed, and the field `mctx` is misleading since
  `synthInstance` methods operate over different `MetavarContext`s simultaneously.
  That being said, we still use `extends` because it makes it simpler to move from
  `M` to `MetaM`.
-/
structure State where
  result         : Option Expr                    := none
  generatorStack : Array GeneratorNode           := #[]
  resumeStack    : Array (ConsumerNode × Answer) := #[]
  tableEntries   : HashMap Expr TableEntry       := {}

abbrev SynthM := ReaderT Context $ StateRefT State MetaM

def checkMaxHeartbeats : SynthM Unit := do
  Core.checkMaxHeartbeatsCore "typeclass" `synthInstance.maxHeartbeats (← read).maxHeartbeats

@[inline] def mapMetaM (f : forall {α}, MetaM α → MetaM α) {α} : SynthM α → SynthM α :=
  monadMap @f

instance {α} : Inhabited (SynthM α) where
  default := fun _ _ => arbitrary

/-- Return globals and locals instances that may unify with `type` -/
def getInstances (type : Expr) : MetaM (Array Expr) := do
  -- We must retrieve `localInstances` before we use `forallTelescopeReducing` because it will update the set of local instances
  let localInstances ← getLocalInstances
  forallTelescopeReducing type fun _ type => do
    let className? ← isClass? type
    match className? with
    | none   => throwError $ "type class instance expected" ++ indentExpr type
    | some className =>
      let globalInstances ← getGlobalInstancesIndex
      let result ← globalInstances.getUnify type
      -- Using insertion sort because it is stable and the array `result` should be mostly sorted.
      -- Most instances have default priority.
      let result := result.insertionSort fun e₁ e₂ => e₁.priority < e₂.priority
      let result ← result.mapM fun e => match e.val with
        | Expr.const constName us _ => return e.val.updateConst! (← us.mapM (fun _ => mkFreshLevelMVar))
        | _ => panic! "global instance is not a constant"
      trace[Meta.synthInstance.globalInstances]! "{type}, {result}"
      let result := localInstances.foldl (init := result) fun (result : Array Expr) linst =>
```

```
      if linst.className == className then result.push linst.fvar else result
    pure result

def mkGeneratorNode? (key mvar : Expr) : MetaM (Option GeneratorNode) := do
  let mvarType  ← inferType mvar
  let mvarType  ← instantiateMVars mvarType
  let instances ← getInstances mvarType
  if instances.isEmpty then
    pure none
  else
    let mctx ← getMCtx
    pure $ some {
      mvar            := mvar,
      key             := key,
      mctx            := mctx,
      instances       := instances,
      currInstanceIdx := instances.size
    }

/-- Create a new generator node for `mvar` and add `waiter` as its waiter.
    `key` must be `mkTableKey mctx mvarType`. -/
def newSubgoal (mctx : MetavarContext) (key : Expr) (mvar : Expr) (waiter : Waiter) : SynthM Unit :=
  withMCtx mctx do
    trace[Meta.synthInstance.newSubgoal]! key
    match (← mkGeneratorNode? key mvar) with
    | none      => pure ()
    | some node =>
      let entry : TableEntry := { waiters := #[waiter] }
      modify fun s =>
       { s with
         generatorStack := s.generatorStack.push node,
         tableEntries   := s.tableEntries.insert key entry }

def findEntry? (key : Expr) : SynthM (Option TableEntry) := do
  return (← get).tableEntries.find? key

def getEntry (key : Expr) : SynthM TableEntry := do
  match (← findEntry? key) with
  | none       => panic! "invalid key at synthInstance"
  | some entry => pure entry

/--
  Create a `key` for the goal associated with the given metavariable.
  That is, we create a key for the type of the metavariable.

  We must instantiate assigned metavariables before we invoke `mkTableKey`. -/
def mkTableKeyFor (mctx : MetavarContext) (mvar : Expr) : SynthM Expr :=
```

```
  withMCtx mctx do
    let mvarType ← inferType mvar
    let mvarType ← instantiateMVars mvarType
    return mkTableKey mctx mvarType

/- See `getSubgoals` and `getSubgoalsAux`

   We use the parameter `j` to reduce the number of `instantiate*` invocations.
   It is the same approach we use at `forallTelescope` and `lambdaTelescope`.
   Given `getSubgoalsAux args j subgoals instVal type`,
   we have that `type.instantiateRevRange j args.size args` does not have loose bound variables. -/
structure SubgoalsResult where
  subgoals     : List Expr
  instVal      : Expr
  instTypeBody : Expr

private partial def getSubgoalsAux (lctx : LocalContext) (localInsts : LocalInstances) (xs : Array Expr)
    : Array Expr → Nat → List Expr → Expr → Expr → MetaM SubgoalsResult
  | args, j, subgoals, instVal, Expr.forallE n d b c => do
    let d        := d.instantiateRevRange j args.size args
    let mvarType ← mkForallFVars xs d
    let mvar     ← mkFreshExprMVarAt lctx localInsts mvarType
    let arg      := mkAppN mvar xs
    let instVal  := mkApp instVal arg
    let subgoals := if c.binderInfo.isInstImplicit then mvar::subgoals else subgoals
    let args     := args.push (mkAppN mvar xs)
    getSubgoalsAux lctx localInsts xs args j subgoals instVal b
  | args, j, subgoals, instVal, type => do
    let type := type.instantiateRevRange j args.size args
    let type ← whnf type
    if type.isForall then
      getSubgoalsAux lctx localInsts xs args args.size subgoals instVal type
    else
      pure ⟨subgoals, instVal, type⟩

/--
  `getSubgoals lctx localInsts xs inst` creates the subgoals for the instance `inst`.
  The subgoals are in the context of the free variables `xs`, and
  `(lctx, localInsts)` is the local context and instances before we added the free variables to it.

  This extra complication is required because
    1- We want all metavariables created by `synthInstance` to share the same local context.
    2- We want to ensure that applications such as `mvar xs` are higher order patterns.

  The method `getGoals` create a new metavariable for each parameter of `inst`.
  For example, suppose the type of `inst` is `forall (x_1 : A_1) ... (x_n : A_n), B x_1 ... x_n`.
  Then, we create the metavariables `?m_i : forall xs, A_i`, and return the subset of these
```

```
  metavariables that are instance implicit arguments, and the expressions:
    - `inst (?m_1 xs) ... (?m_n xs)` (aka `instVal`)
    - `B (?m_1 xs) ... (?m_n xs)` -/
def getSubgoals (lctx : LocalContext) (localInsts : LocalInstances) (xs : Array Expr) (inst : Expr) : MetaM SubgoalsResult := do
  let instType ← inferType inst
  let result ← getSubgoalsAux lctx localInsts xs #[] 0 [] inst instType
  match inst.getAppFn with
  | Expr.const constName _ _ =>
    let env ← getEnv
    if hasInferTCGoalsRLAttribute env constName then
      pure result
    else
      pure { result with subgoals := result.subgoals.reverse }
  | _ => pure result

def tryResolveCore (mvar : Expr) (inst : Expr) : MetaM (Option (MetavarContext × List Expr)) := do
  let mvarType   ← inferType mvar
  let lctx       ← getLCtx
  let localInsts ← getLocalInstances
  forallTelescopeReducing mvarType fun xs mvarTypeBody => do
    let ⟨subgoals, instVal, instTypeBody⟩ ← getSubgoals lctx localInsts xs inst
    trace[Meta.synthInstance.tryResolve]! "{mvarTypeBody} =?= {instTypeBody}"
    if (← isDefEq mvarTypeBody instTypeBody) then
      let instVal ← mkLambdaFVars xs instVal
      if (← isDefEq mvar instVal) then
        trace[Meta.synthInstance.tryResolve]! "success"
        pure (some ((← getMCtx), subgoals))
      else
        trace[Meta.synthInstance.tryResolve]! "failure assigning"
        pure none
    else
      trace[Meta.synthInstance.tryResolve]! "failure"
      pure none

/--
  Try to synthesize metavariable `mvar` using the instance `inst`.
  Remark: `mctx` contains `mvar`.
  If it succeeds, the result is a new updated metavariable context and a new list of subgoals.
  A subgoal is created for each instance implicit parameter of `inst`. -/
def tryResolve (mctx : MetavarContext) (mvar : Expr) (inst : Expr) : SynthM (Option (MetavarContext × List Expr)) :=
  traceCtx `Meta.synthInstance.tryResolve <| withMCtx mctx <| tryResolveCore mvar inst

/--
  Assign a precomputed answer to `mvar`.
  If it succeeds, the result is a new updated metavariable context and a new list of subgoals. -/
def tryAnswer (mctx : MetavarContext) (mvar : Expr) (answer : Answer) : SynthM (Option MetavarContext) :=
  withMCtx mctx do
```

```
    let (_, _, val) ← openAbstractMVarsResult answer.result
    if (← isDefEq mvar val) then
      pure (some (← getMCtx))
    else
      pure none

/-- Move waiters that are waiting for the given answer to the resume stack. -/
def wakeUp (answer : Answer) : Waiter → SynthM Unit
  | Waiter.root               => do
    if answer.result.paramNames.isEmpty && answer.result.numMVars == 0 then
      modify fun s => { s with result := answer.result.expr }
    else
      let (_, _, answerExpr) ← openAbstractMVarsResult answer.result
      trace[Meta.synthInstance]! "skip answer containing metavariables {answerExpr}"
      pure ()
  | Waiter.consumerNode cNode =>
    modify fun s => { s with resumeStack := s.resumeStack.push (cNode, answer) }

def isNewAnswer (oldAnswers : Array Answer) (answer : Answer) : Bool :=
  oldAnswers.all fun oldAnswer => do
    -- Remark: isDefEq here is too expensive. TODO: if `==` is too imprecise, add some light normalization to `resultType` at `addAnswer`
    -- iseq ← isDefEq oldAnswer.resultType answer.resultType; pure (!iseq)
    oldAnswer.resultType != answer.resultType

private def mkAnswer (cNode : ConsumerNode) : MetaM Answer :=
  withMCtx cNode.mctx do
    traceM `Meta.synthInstance.newAnswer do m!"size: {cNode.size}, {← inferType cNode.mvar}"
    let val ← instantiateMVars cNode.mvar
    trace[Meta.synthInstance.newAnswer]! "val: {val}"
    let result ← abstractMVars val -- assignable metavariables become parameters
    let resultType ← inferType result.expr
    pure { result := result, resultType := resultType, size := cNode.size + 1 }

/--
  Create a new answer after `cNode` resolved all subgoals.
  That is, `cNode.subgoals == []`.
  And then, store it in the tabled entries map, and wakeup waiters. -/
def addAnswer (cNode : ConsumerNode) : SynthM Unit := do
  if cNode.size ≥ (← read).maxResultSize then
    traceM `Meta.synthInstance.discarded do m!"size: {cNode.size} ≥ {(← read).maxResultSize}, {← inferType cNode.mvar}"
    return ()
  else
    let answer ← mkAnswer cNode
    -- Remark: `answer` does not contain assignable or assigned metavariables.
    let key := cNode.key
    let entry ← getEntry key
```

```
    if isNewAnswer entry.answers answer then
      let newEntry := { entry with answers := entry.answers.push answer }
      modify fun s => { s with tableEntries := s.tableEntries.insert key newEntry }
      entry.waiters.forM (wakeUp answer)

/-- Process the next subgoal in the given consumer node. -/
def consume (cNode : ConsumerNode) : SynthM Unit :=
  match cNode.subgoals with
  | []      => addAnswer cNode
  | mvar::_ => do
     let waiter := Waiter.consumerNode cNode
     let key ← mkTableKeyFor cNode.mctx mvar
     let entry? ← findEntry? key
     match entry? with
     | none       => newSubgoal cNode.mctx key mvar waiter
     | some entry => modify fun s =>
       { s with
         resumeStack  := entry.answers.foldl (fun s answer => s.push (cNode, answer)) s.resumeStack,
         tableEntries := s.tableEntries.insert key { entry with waiters := entry.waiters.push waiter } }

def getTop : SynthM GeneratorNode := do
  pure (← get).generatorStack.back

@[inline] def modifyTop (f : GeneratorNode → GeneratorNode) : SynthM Unit :=
  modify fun s => { s with generatorStack := s.generatorStack.modify (s.generatorStack.size - 1) f }

/-- Try the next instance in the node on the top of the generator stack. -/
def generate : SynthM Unit := do
  let gNode ← getTop
  if gNode.currInstanceIdx == 0  then
    modify fun s => { s with generatorStack := s.generatorStack.pop }
  else do
    let key  := gNode.key
    let idx  := gNode.currInstanceIdx - 1
    let inst := gNode.instances.get! idx
    let mctx := gNode.mctx
    let mvar := gNode.mvar
    trace[Meta.synthInstance.generate]! "instance {inst}"
    modifyTop fun gNode => { gNode with currInstanceIdx := idx }
    match (← tryResolve mctx mvar inst) with
    | none                  => pure ()
    | some (mctx, subgoals) => consume { key := key, mvar := mvar, subgoals := subgoals, mctx := mctx, size := 0 }

def getNextToResume : SynthM (ConsumerNode × Answer) := do
  let s ← get
  let r := s.resumeStack.back
  modify fun s => { s with resumeStack := s.resumeStack.pop }
```

```
  pure r

/--
  Given `(cNode, answer)` on the top of the resume stack, continue execution by using `answer` to solve the
  next subgoal. -/
def resume : SynthM Unit := do
  let (cNode, answer) ← getNextToResume
  match cNode.subgoals with
  | []         => panic! "resume found no remaining subgoals"
  | mvar::rest =>
    match (← tryAnswer cNode.mctx mvar answer) with
    | none      => pure ()
    | some mctx =>
      withMCtx mctx <| traceM `Meta.synthInstance.resume do
        let goal    ← inferType cNode.mvar
        let subgoal ← inferType mvar
        pure m!"size: {cNode.size + answer.size}, {goal} <== {subgoal}"
      consume { key := cNode.key, mvar := cNode.mvar, subgoals := rest, mctx := mctx, size := cNode.size + answer.size }

def step : SynthM Bool := do
  checkMaxHeartbeats
  let s ← get
  if !s.resumeStack.isEmpty then
    resume
    pure true
  else if !s.generatorStack.isEmpty then
    generate
    pure true
  else
    pure false

def getResult : SynthM (Option Expr) := do
  pure (← get).result

partial def synth : SynthM (Option Expr) := do
  if (← step) then
    match (← getResult) with
    | none        => synth
    | some result => pure result
  else
    trace[Meta.synthInstance]! "failed"
    pure none

def main (type : Expr) (maxResultSize : Nat) : MetaM (Option Expr) :=
  withCurrHeartbeats <| traceCtx `Meta.synthInstance do
     trace[Meta.synthInstance]! "main goal {type}"
     let mvar ← mkFreshExprMVar type
```

```
    let mctx ← getMCtx
    let key    := mkTableKey mctx type
    let action : SynthM (Option Expr) := do
      newSubgoal mctx key mvar Waiter.root
      synth
    action.run { maxResultSize := maxResultSize, maxHeartbeats := getMaxHeartbeats (← getOptions) } |>.run' {}

end SynthInstance

/-
Type class parameters can be annotated with `outParam` annotations.

Given `C a_1 ... a_n`, we replace `a_i` with a fresh metavariable `?m_i` IF
`a_i` is an `outParam`.
The result is type correct because we reject type class declarations IF
it contains a regular parameter X that depends on an `out` parameter Y.

Then, we execute type class resolution as usual.
If it succeeds, and metavariables ?m_i have been assigned, we try to unify
the original type `C a_1 ... a_n` witht the normalized one.
-/

private def preprocess (type : Expr) : MetaM Expr :=
  forallTelescopeReducing type fun xs type => do
    let type ← whnf type
    mkForallFVars xs type

private def preprocessLevels (us : List Level) : MetaM (List Level × Bool) := do
  let mut r := #[]
  let mut modified := false
  for u in us do
    let u ← instantiateLevelMVars u
    if u.hasMVar then
      r := r.push (← mkFreshLevelMVar)
      modified := true
    else
      r := r.push u
  return (r.toList, modified)

private partial def preprocessArgs (type : Expr) (i : Nat) (args : Array Expr) : MetaM (Array Expr) := do
  if h : i < args.size then
    let type ← whnf type
    match type with
    | Expr.forallE _ d b _ => do
      let arg := args.get ⟨i, h⟩
      let arg ← if isOutParam d then mkFreshExprMVar d else pure arg
      let args := args.set ⟨i, h⟩ arg
```

```
      preprocessArgs (b.instantiate1 arg) (i+1) args
    | _ =>
      throwError "type class resolution failed, insufficient number of arguments" -- TODO improve error message
  else
    return args

private def preprocessOutParam (type : Expr) : MetaM Expr :=
  forallTelescope type fun xs typeBody => do
    match typeBody.getAppFn with
    | c@(Expr.const constName us _) =>
      let env ← getEnv
      if !hasOutParams env constName then
        /- We treat all universe level parameters as "outParam" -/
        let (us, modified) ← preprocessLevels us
        if modified then
          let c := mkConst constName us
          mkForallFVars xs (mkAppN c typeBody.getAppArgs)
        else
          return type
      else do
        let args := typeBody.getAppArgs
        let (us, _) ← preprocessLevels us
        let c := mkConst constName us
        let cType ← inferType c
        let args ← preprocessArgs cType 0 args
        mkForallFVars xs (mkAppN c args)
    | _ =>
      return type

/-
  Remark: when `maxResultSize? == none`, the configuration option `synthInstance.maxResultSize` is used.
  Remark: we use a different option for controlling the maximum result size for coercions.
-/

def synthInstance? (type : Expr) (maxResultSize? : Option Nat := none) : MetaM (Option Expr) := do profileitM Exception "typeclass inference" (← getOptions) do
  let opts ← getOptions
  let maxResultSize := maxResultSize?.getD (synthInstance.maxSize.get opts)
  let inputConfig ← getConfig
  withConfig (fun config => { config with isDefEqStuckEx := true, transparency := TransparencyMode.instances,
                                        foApprox := true, ctxApprox := true, constApprox := false }) do
    let type ← instantiateMVars type
    let type ← preprocess type
    let s ← get
    match s.cache.synthInstance.find? type with
    | some result => pure result
    | none        =>
```

```
    let result? ← withNewMCtxDepth do
      let normType ← preprocessOutParam type
      trace[Meta.synthInstance]! "{type} ==> {normType}"
      match (← SynthInstance.main normType maxResultSize) with
      | none        => pure none
      | some result =>
        trace[Meta.synthInstance]! "FOUND result {result}"
        let result ← instantiateMVars result
        if (← hasAssignableMVar result) then
          trace[Meta.synthInstance]! "Failed has assignable mvar {result.setOption `pp.all true}"
          pure none
        else
          pure (some result)
    let result? ← match result? with
      | none        => pure none
      | some result => do
        trace[Meta.synthInstance]! "result {result}"
        let resultType ← inferType result
        if (← withConfig (fun _ => inputConfig) <| isDefEq type resultType) then
          let result ← instantiateMVars result
          pure (some result)
        else
          trace[Meta.synthInstance]! "result type{indentExpr resultType}\nis not definitionally equal to{indentExpr type}"
          pure none
    if type.hasMVar then
      pure result?
    else do
      modify fun s => { s with cache := { s.cache with synthInstance := s.cache.synthInstance.insert type result? } }
      pure result?

/--
  Return `LOption.some r` if succeeded, `LOption.none` if it failed, and `LOption.undef` if
  instance cannot be synthesized right now because `type` contains metavariables. -/
def trySynthInstance (type : Expr) (maxResultSize? : Option Nat := none) : MetaM (LOption Expr) := do
  catchInternalId isDefEqStuckExceptionId
    (toLOptionM <| synthInstance? type maxResultSize?)
    (fun _ => pure LOption.undef)

def synthInstance (type : Expr) (maxResultSize? : Option Nat := none) : MetaM Expr :=
  catchInternalId isDefEqStuckExceptionId
    (do
      let result? ← synthInstance? type maxResultSize?
      match result? with
      | some result => pure result
      | none        => throwError! "failed to synthesize{indentExpr type}")
    (fun _ => throwError! "failed to synthesize{indentExpr type}")
```

```
private def synthPendingImp (mvarId : MVarId) (maxResultSize? : Option Nat) : MetaM Bool := do
  let mvarDecl ← getMVarDecl mvarId
  match mvarDecl.kind with
  | MetavarKind.synthetic =>
    match (← isClass? mvarDecl.type) with
    | none   => pure false
    | some _ => do
      let val? ← catchInternalId isDefEqStuckExceptionId (synthInstance? mvarDecl.type maxResultSize?) (fun _ => pure none)
      match val? with
      | none     => pure false
      | some val =>
        if (← isExprMVarAssigned mvarId) then
          pure false
        else
          assignExprMVar mvarId val
          pure true
  | _ => pure false

builtin_initialize
  synthPendingRef.set (synthPendingImp · none)

builtin_initialize
  registerTraceClass `Meta.synthInstance
  registerTraceClass `Meta.synthInstance.globalInstances
  registerTraceClass `Meta.synthInstance.newSubgoal
  registerTraceClass `Meta.synthInstance.tryResolve
  registerTraceClass `Meta.synthInstance.resume
  registerTraceClass `Meta.synthInstance.generate

end Lean.Meta
// ::::::::::::::
// Tactic.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.Tactic.Intro
import Lean.Meta.Tactic.Assumption
import Lean.Meta.Tactic.Contradiction
import Lean.Meta.Tactic.Apply
import Lean.Meta.Tactic.Revert
import Lean.Meta.Tactic.Clear
import Lean.Meta.Tactic.Assert
import Lean.Meta.Tactic.Rewrite
import Lean.Meta.Tactic.Generalize
```

```
import Lean.Meta.Tactic.Replace
import Lean.Meta.Tactic.Induction
import Lean.Meta.Tactic.Cases
import Lean.Meta.Tactic.ElimInfo
import Lean.Meta.Tactic.Delta
import Lean.Meta.Tactic.Constructor
import Lean.Meta.Tactic.Simp
// ::::::::::::::
// Transform.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.Meta.Basic

namespace Lean

inductive TransformStep where
  | done  (e : Expr)
  | visit (e : Expr)

namespace Core

/--
  Tranform the expression `input` using `pre` and `post`.
  - `pre s` is invoked before visiting the children of subterm 's'. If the result is `TransformStep.visit sNew`, then
     `sNew` is traversed by transform. If the result is `TransformStep.visit sNew`, then `s` is just replaced with `sNew`.
     In both cases, `sNew` must be definitionally equal to `s`
  - `post s` is invoked after visiting the children of subterm `s`.

  The term `s` in both `pre s` and `post s` may contain loose bound variables. So, this method is not appropriate for
  if one needs to apply operations (e.g., `whnf`, `inferType`) that do not handle loose bound variables.
  Consider using `Meta.transform` to avoid loose bound variables.

  This method is useful for applying transformations such as beta-reduction and delta-reduction.
-/
partial def transform {m} [Monad m] [MonadLiftT CoreM m] [MonadControlT CoreM m]
    (input : Expr)
    (pre   : Expr → m TransformStep := fun e => return TransformStep.visit e)
    (post  : Expr → m TransformStep := fun e => return TransformStep.done e)
    : m Expr :=
  let inst : STWorld IO.RealWorld m := ⟨⟩
  let inst : MonadLiftT (ST IO.RealWorld) m := { monadLift := fun x => liftM (m := CoreM) (liftM (m := ST IO.RealWorld) x) }
  let rec visit (e : Expr) : MonadCacheT Expr Expr m Expr :=
    checkCache e fun _ => Core.withIncRecDepth do
```

```
      let rec visitPost (e : Expr) : MonadCacheT Expr Expr m Expr := do
        match (← post e) with
        | TransformStep.done e  => pure e
        | TransformStep.visit e => visit e
      match (← pre e) with
      | TransformStep.done e  => pure e
      | TransformStep.visit e => match e with
        | Expr.forallE _ d b _ => visitPost (e.updateForallE! (← visit d) (← visit b))
        | Expr.lam _ d b _     => visitPost (e.updateLambdaE! (← visit d) (← visit b))
        | Expr.letE _ t v b _  => visitPost (e.updateLet! (← visit t) (← visit v) (← visit b))
        | Expr.app ..          => e.withApp fun f args => do visitPost (mkAppN (← visit f) (← args.mapM visit))
        | Expr.mdata _ b _     => visitPost (e.updateMData! (← visit b))
        | Expr.proj _ _ b _    => visitPost (e.updateProj! (← visit b))
        | _                    => visitPost e
  visit input |>.run

def betaReduce (e : Expr) : CoreM Expr :=
  transform e (pre := fun e => return TransformStep.visit e.headBeta)

end Core

namespace Meta

/--
  Similar to `Core.transform`, but terms provided to `pre` and `post` do not contain loose bound variables.
  So, it is safe to use any `MetaM` method at `pre` and `post`. -/
partial def transform {m} [Monad m] [MonadLiftT MetaM m] [MonadControlT MetaM m]
    (input : Expr)
    (pre   : Expr → m TransformStep := fun e => return TransformStep.visit e)
    (post  : Expr → m TransformStep := fun e => return TransformStep.done e)
    : m Expr :=
  let inst : STWorld IO.RealWorld m := ⟨⟩
  let inst : MonadLiftT (ST IO.RealWorld) m := { monadLift := fun x => liftM (m := MetaM) (liftM (m := ST IO.RealWorld) x) }
  let rec visit (e : Expr) : MonadCacheT Expr Expr m Expr :=
    checkCache e fun _ => Meta.withIncRecDepth do
      let rec visitPost (e : Expr) : MonadCacheT Expr Expr m Expr := do
        match (← post e) with
        | TransformStep.done e  => pure e
        | TransformStep.visit e => visit e
      let rec visitLambda (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
        match e with
        | Expr.lam n d b c =>
          withLocalDecl n c.binderInfo (← visit (d.instantiateRev fvars)) fun x =>
            visitLambda (fvars.push x) b
        | e => visitPost (← mkLambdaFVars fvars (← visit (e.instantiateRev fvars)))
      let rec visitForall (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
        match e with
```

```
        | Expr.forallE n d b c =>
          withLocalDecl n c.binderInfo (← visit (d.instantiateRev fvars)) fun x =>
            visitForall (fvars.push x) b
        | e => visitPost (← mkForallFVars fvars (← visit (e.instantiateRev fvars)))
      let rec visitLet (fvars : Array Expr) (e : Expr) : MonadCacheT Expr Expr m Expr := do
        match e with
        | Expr.letE n t v b _ =>
          withLetDecl n (← visit (t.instantiateRev fvars)) (← visit (v.instantiateRev fvars)) fun x =>
            visitLet (fvars.push x) b
        | e => visitPost (← mkLetFVars fvars (← visit (e.instantiateRev fvars)))
      let visitApp (e : Expr) : MonadCacheT Expr Expr m Expr :=
        e.withApp fun f args => do
          visitPost (mkAppN (← visit f) (← args.mapM visit))
      match (← pre e) with
      | TransformStep.done e  => pure e
      | TransformStep.visit e => match e with
        | Expr.forallE ..    => visitForall #[] e
        | Expr.lam ..        => visitLambda #[] e
        | Expr.letE ..       => visitLet #[] e
        | Expr.app ..        => visitApp e
        | Expr.mdata _ b _   => visitPost (e.updateMData! (← visit b))
        | Expr.proj _ _ b _  => visitPost (e.updateProj! (← visit b))
        | _                  => visitPost e
  visit input |>.run

def zetaReduce (e : Expr) : MetaM Expr := do
  let lctx ← getLCtx
  let pre (e : Expr) : CoreM TransformStep := do
    match e with
    | Expr.fvar fvarId _ =>
      match lctx.find? fvarId with
      | none => return TransformStep.done e
      | some localDecl =>
        if let some value := localDecl.value? then
          return TransformStep.visit value
        else
          return TransformStep.done e
    | e => if e.hasFVar then return TransformStep.visit e else return TransformStep.done e
  liftM (m := CoreM) <| Core.transform e (pre := pre)

end Meta
end Lean
// ::::::::::::::
// TransparencyMode.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
```

```
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
namespace Lean.Meta

inductive TransparencyMode where
  | all | default | reducible | instances
  deriving Inhabited, BEq, Repr

namespace TransparencyMode

def hash : TransparencyMode → USize
  | all       => 7
  | default   => 11
  | reducible => 13
  | instances => 17

instance : Hashable TransparencyMode := ⟨hash⟩

def lt : TransparencyMode → TransparencyMode → Bool
  | reducible, default   => true
  | reducible, all       => true
  | reducible, instances => true
  | instances, default   => true
  | instances, all       => true
  | default,   all       => true
  | _,         _         => false

end TransparencyMode

end Lean.Meta
// ::::::::::::::
// UnificationHint.lean
// ::::::::::::::
/-
Copyright (c) 2020 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.ScopedEnvExtension
import Lean.Util.Recognizers
import Lean.Meta.DiscrTree
import Lean.Meta.LevelDefEq
import Lean.Meta.SynthInstance

namespace Lean.Meta

```

```
structure UnificationHintEntry where
  keys        : Array DiscrTree.Key
  val         : Name
  deriving Inhabited

structure UnificationHints where
  discrTree       : DiscrTree Name := DiscrTree.empty
  deriving Inhabited

instance : ToFormat UnificationHints where
  format h := fmt h.discrTree

def UnificationHints.add (hints : UnificationHints) (e : UnificationHintEntry) : UnificationHints :=
  { hints with discrTree := hints.discrTree.insertCore e.keys e.val }

builtin_initialize unificationHintExtension : SimpleScopedEnvExtension UnificationHintEntry UnificationHints ←
  registerSimpleScopedEnvExtension {
    name     := `unifHints
    addEntry := UnificationHints.add
    initial  := {}
  }

structure UnificationConstraint where
  lhs : Expr
  rhs : Expr

structure UnificationHint where
  pattern     : UnificationConstraint
  constraints : List UnificationConstraint

private partial def decodeUnificationHint (e : Expr) : ExceptT MessageData Id UnificationHint := do
  decode e #[]
where
  decodeConstraint (e : Expr) : ExceptT MessageData Id UnificationConstraint :=
    match e.eq? with
    | some (_, lhs, rhs) => return UnificationConstraint.mk lhs rhs
    | none => throw m!"invalid unification hint constraint, unexpected term{indentExpr e}"
  decode (e : Expr) (cs : Array UnificationConstraint) : ExceptT MessageData Id UnificationHint := do
    match e with
    | Expr.forallE _ d b _ => do
      let c ← decodeConstraint d
      if b.hasLooseBVars then
        throw m!"invalid unification hint constraint, unexpected dependency{indentExpr e}"
      decode b (cs.push c)
    | _ => do
      let p ← decodeConstraint e
      return { pattern := p, constraints := cs.toList }
```

```
private partial def validateHint (declName : Name) (hint : UnificationHint) : MetaM Unit := do
  hint.constraints.forM fun c => do
    unless (← isDefEq c.lhs c.rhs) do
      throwError! "invalid unification hint, failed to unify constraint left-hand-side{indentExpr c.lhs}\nwith right-hand-side{indentExpr c.rhs}"
  unless (← isDefEq hint.pattern.lhs hint.pattern.rhs) do
    throwError! "invalid unification hint, failed to unify pattern left-hand-side{indentExpr hint.pattern.lhs}\nwith right-hand-side{indentExpr hint.pattern.rhs}"

def addUnificationHint (declName : Name) (kind : AttributeKind) : MetaM Unit :=
  withNewMCtxDepth do
    let info ← getConstInfo declName
    match info.value? with
    | none => throwError! "invalid unification hint, it must be a definition"
    | some val =>
      let (_, _, body) ← lambdaMetaTelescope val
      match decodeUnificationHint body with
      | Except.error msg => throwError msg
      | Except.ok hint =>
        let keys ← DiscrTree.mkPath hint.pattern.lhs
        validateHint declName hint
        unificationHintExtension.add { keys := keys, val := declName } kind
        trace[Meta.debug]! "addUnificationHint: {unificationHintExtension.getState (← getEnv)}"

builtin_initialize
  registerBuiltinAttribute {
    name  := `unificationHint
    descr := "unification hint"
    add   := fun declName stx kind => do
      Attribute.Builtin.ensureNoArgs stx
      discard <| addUnificationHint declName kind |>.run
  }

def tryUnificationHints (t s : Expr) : MetaM Bool := do
  trace[Meta.isDefEq.hint]! "{t} =?= {s}"
  unless (← read).config.unificationHints do
    return false
  if t.isMVar then
    return false
  let hints := unificationHintExtension.getState (← getEnv)
  let candidates ← hints.discrTree.getMatch t
  for candidate in candidates do
    if (← tryCandidate candidate) then
      return true
  return false
where
```

```
  isDefEqPattern p e :=
    withReducible <| Meta.isExprDefEqAux p e

  tryCandidate candidate : MetaM Bool :=
    traceCtx `Meta.isDefEq.hint <| commitWhen do
      trace[Meta.isDefEq.hint]! "trying hint {candidate} at {t} =?= {s}"
      let cinfo ← getConstInfo candidate
      let us ← cinfo.levelParams.mapM fun _ => mkFreshLevelMVar
      let val := cinfo.instantiateValueLevelParams us
      let (xs, bis, body) ← lambdaMetaTelescope val
      let hint? ← withConfig (fun cfg => { cfg with unificationHints := false }) do
        match decodeUnificationHint body with
        | Except.error _ => return none
        | Except.ok hint =>
          if (← isDefEqPattern hint.pattern.lhs t <&&> isDefEqPattern hint.pattern.rhs s) then
            return some hint
          else
            return none
      match hint? with
      | none      => return false
      | some hint =>
        trace[Meta.isDefEq.hint]! "{candidate} succeeded, applying constraints"
        for c in hint.constraints do
          unless (← Meta.isExprDefEqAux c.lhs c.rhs) do
            return false
        for x in xs, bi in bis do
          if bi == BinderInfo.instImplicit then
            match (← trySynthInstance (← inferType x)) with
            | LOption.some val => unless (← isDefEq x val) do return false
            | _                => return false
        return true

builtin_initialize
  registerTraceClass `Meta.isDefEq.hint

end Lean.Meta
// ::::::::::::::
// WHNF.lean
// ::::::::::::::
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Leonardo de Moura
-/
import Lean.ToExpr
import Lean.AuxRecursor
import Lean.ProjFns
```

```
import Lean.Meta.Basic
import Lean.Meta.LevelDefEq
import Lean.Meta.GetConst
import Lean.Meta.Match.MatcherInfo

namespace Lean.Meta

/- ===========================
   Smart unfolding support
   =========================== -/

def smartUnfoldingSuffix := "_sunfold"

@[inline] def mkSmartUnfoldingNameFor (declName : Name) : Name :=
  Name.mkStr declName smartUnfoldingSuffix

register_builtin_option smartUnfolding : Bool := {
  defValue := true
  descr := "when computing weak head normal form, use auxiliary definition created for functions defined by structural recursion"
}

/- ===========================
   Helper methods
   =========================== -/
def isAuxDef (constName : Name) : MetaM Bool := do
  let env ← getEnv
  return isAuxRecursor env constName || isNoConfusion env constName

@[inline] private def matchConstAux {α} (e : Expr) (failK : Unit → MetaM α) (k : ConstantInfo → List Level → MetaM α) : MetaM α :=
  match e with
  | Expr.const name lvls _ => do
    let (some cinfo) ← getConst? name | failK ()
    k cinfo lvls
  | _ => failK ()

/- ===========================
   Helper functions for reducing recursors
   =========================== -/

private def getFirstCtor (d : Name) : MetaM (Option Name) := do
  let some (ConstantInfo.inductInfo { ctors := ctor::_, ..}) ← getConstNoEx? d | pure none
  return some ctor

private def mkNullaryCtor (type : Expr) (nparams : Nat) : MetaM (Option Expr) :=
  match type.getAppFn with
  | Expr.const d lvls _ => do
    let (some ctor) ← getFirstCtor d | pure none
```

```
    return mkAppN (mkConst ctor lvls) (type.getAppArgs.shrink nparams)
  | _ =>
    return none

def toCtorIfLit : Expr → Expr
  | Expr.lit (Literal.natVal v) _ =>
    if v == 0 then mkConst `Nat.zero
    else mkApp (mkConst `Nat.succ) (mkNatLit (v-1))
  | Expr.lit (Literal.strVal v) _ =>
    mkApp (mkConst `String.mk) (toExpr v.toList)
  | e => e

private def getRecRuleFor (recVal : RecursorVal) (major : Expr) : Option RecursorRule :=
  match major.getAppFn with
  | Expr.const fn _ _ => recVal.rules.find? fun r => r.ctor == fn
  | _                 => none

private def toCtorWhenK (recVal : RecursorVal) (major : Expr) : MetaM (Option Expr) := do
  let majorType ← inferType major
  let majorType ← whnf majorType
  let majorTypeI := majorType.getAppFn
  if !majorTypeI.isConstOf recVal.getInduct then
    return none
  else if majorType.hasExprMVar && majorType.getAppArgs[recVal.numParams:].any Expr.hasExprMVar then
    return none
  else do
    let (some newCtorApp) ← mkNullaryCtor majorType recVal.numParams | pure none
    let newType ← inferType newCtorApp
    if (← isDefEq majorType newType) then
      return newCtorApp
    else
      return none

/-- Auxiliary function for reducing recursor applications. -/
private def reduceRec {α} (recVal : RecursorVal) (recLvls : List Level) (recArgs : Array Expr) (failK : Unit → MetaM α) (successK : Expr → MetaM α) : MetaM α :=
  let majorIdx := recVal.getMajorIdx
  if h : majorIdx < recArgs.size then do
    let major := recArgs.get ⟨majorIdx, h⟩
    let mut major ← whnf major
    if recVal.k then
      let newMajor ← toCtorWhenK recVal major
      major := newMajor.getD major
    let major := toCtorIfLit major
    match getRecRuleFor recVal major with
    | some rule =>
      let majorArgs := major.getAppArgs
```

```
      if recLvls.length != recVal.levelParams.length then
        failK ()
      else
        let rhs := rule.rhs.instantiateLevelParams recVal.levelParams recLvls
        -- Apply parameters, motives and minor premises from recursor application.
        let rhs := mkAppRange rhs 0 (recVal.numParams+recVal.numMotives+recVal.numMinors) recArgs
        /- The number of parameters in the constructor is not necessarily
           equal to the number of parameters in the recursor when we have
           nested inductive types. -/
        let nparams := majorArgs.size - rule.nfields
        let rhs := mkAppRange rhs nparams majorArgs.size majorArgs
        let rhs := mkAppRange rhs (majorIdx + 1) recArgs.size recArgs
        successK rhs
    | none => failK ()
  else
    failK ()

/- ===========================
   Helper functions for reducing Quot.lift and Quot.ind
   =========================== -/

/-- Auxiliary function for reducing `Quot.lift` and `Quot.ind` applications. -/
private def reduceQuotRec {α} (recVal  : QuotVal) (recLvls : List Level) (recArgs : Array Expr) (failK : Unit → MetaM α) (successK : Expr → MetaM α) : MetaM α :=
  let process (majorPos argPos : Nat) : MetaM α :=
    if h : majorPos < recArgs.size then do
      let major := recArgs.get ⟨majorPos, h⟩
      let major ← whnf major
      match major with
      | Expr.app (Expr.app (Expr.app (Expr.const majorFn _ _) _ _) _ _) majorArg _ => do
        let some (ConstantInfo.quotInfo { kind := QuotKind.ctor, .. }) ← getConstNoEx? majorFn | failK ()
        let f := recArgs[argPos]
        let r := mkApp f majorArg
        let recArity := majorPos + 1
        successK $ mkAppRange r recArity recArgs.size recArgs
      | _ => failK ()
    else
      failK ()
  match recVal.kind with
  | QuotKind.lift => process 5 3
  | QuotKind.ind  => process 4 3
  | _             => failK ()

/- ===========================
   Helper function for extracting "stuck term"
   =========================== -/
```

```
mutual
  private partial def isRecStuck? (recVal : RecursorVal) (recLvls : List Level) (recArgs : Array Expr) : MetaM (Option MVarId) :=
    if recVal.k then
      -- TODO: improve this case
      return none
    else do
      let majorIdx := recVal.getMajorIdx
      if h : majorIdx < recArgs.size then do
        let major := recArgs.get ⟨majorIdx, h⟩
        let major ← whnf major
        getStuckMVar? major
      else
        return none

  private partial def isQuotRecStuck? (recVal : QuotVal) (recLvls : List Level) (recArgs : Array Expr) : MetaM (Option MVarId) :=
    let process? (majorPos : Nat) : MetaM (Option MVarId) :=
      if h : majorPos < recArgs.size then do
        let major := recArgs.get ⟨majorPos, h⟩
        let major ← whnf major
        getStuckMVar? major
      else
        return none
    match recVal.kind with
    | QuotKind.lift => process? 5
    | QuotKind.ind  => process? 4
    | _             => return none

  /-- Return `some (Expr.mvar mvarId)` if metavariable `mvarId` is blocking reduction. -/
  partial def getStuckMVar? : Expr → MetaM (Option MVarId)
    | Expr.mdata _ e _       => getStuckMVar? e
    | Expr.proj _ _ e _      => do getStuckMVar? (← whnf e)
    | e@(Expr.mvar ..) => do
      let e ← instantiateMVars e
      match e with
      | Expr.mvar mvarId _ => pure (some mvarId)
      | _ => getStuckMVar? e
    | e@(Expr.app f _ _) =>
      let f := f.getAppFn
      match f with
      | Expr.mvar mvarId _       => return some mvarId
      | Expr.const fName fLvls _ => do
        let cinfo? ← getConstNoEx? fName
        match cinfo? with
        | some $ ConstantInfo.recInfo recVal  => isRecStuck? recVal fLvls e.getAppArgs
        | some $ ConstantInfo.quotInfo recVal => isQuotRecStuck? recVal fLvls e.getAppArgs
        | _                                   => return none
      | _ => return none
```

```
    | _ => return none
end

/- ==========================
   Weak Head Normal Form auxiliary combinators
   ========================== -/

/-- Auxiliary combinator for handling easy WHNF cases. It takes a function for handling the "hard" cases as an argument -/
@[specialize] private partial def whnfEasyCases (e : Expr) (k : Expr → MetaM Expr) : MetaM Expr := do
  match e with
  | Expr.forallE ..    => return e
  | Expr.lam ..        => return e
  | Expr.sort ..       => return e
  | Expr.lit ..        => return e
  | Expr.bvar ..       => unreachable!
  | Expr.letE ..       => k e
  | Expr.const ..      => k e
  | Expr.app ..        => k e
  | Expr.proj ..       => k e
  | Expr.mdata _ e _   => whnfEasyCases e k
  | Expr.fvar fvarId _ =>
    let decl ← getLocalDecl fvarId
    match decl with
    | LocalDecl.cdecl .. => return e
    | LocalDecl.ldecl (value := v) (nonDep := nonDep) .. =>
      let cfg ← getConfig
      if nonDep && !cfg.zetaNonDep then
        return e
      else
        when cfg.trackZeta do
          modify fun s => { s with zetaFVarIds := s.zetaFVarIds.insert fvarId }
        whnfEasyCases v k
  | Expr.mvar mvarId _ =>
    match (← getExprMVarAssignment? mvarId) with
    | some v => whnfEasyCases v k
    | none   => return e

/-- Return true iff term is of the form `idRhs ...` -/
private def isIdRhsApp (e : Expr) : Bool :=
  e.isAppOf `idRhs

/-- (@idRhs T f a_1 ... a_n) ==> (f a_1 ... a_n) -/
private def extractIdRhs (e : Expr) : Expr :=
  if !isIdRhsApp e then e
  else
    let args := e.getAppArgs
    if args.size < 2 then e
```

```
    else mkAppRange args[1] 2 args.size args

@[specialize] private def deltaDefinition {α} (c : ConstantInfo) (lvls : List Level)
    (failK : Unit → α) (successK : Expr → α) : α :=
  if c.levelParams.length != lvls.length then failK ()
  else
    let val := c.instantiateValueLevelParams lvls
    successK (extractIdRhs val)

@[specialize] private def deltaBetaDefinition {α} (c : ConstantInfo) (lvls : List Level) (revArgs : Array Expr)
    (failK : Unit → α) (successK : Expr → α) : α :=
  if c.levelParams.length != lvls.length then
    failK ()
  else
    let val := c.instantiateValueLevelParams lvls
    let val := val.betaRev revArgs
    successK (extractIdRhs val)

inductive ReduceMatcherResult where
  | reduced (val : Expr)
  | stuck   (val : Expr)
  | notMatcher
  | partialApp

def reduceMatcher? (e : Expr) : MetaM ReduceMatcherResult := do
  match e.getAppFn with
  | Expr.const declName declLevels _ =>
    let some info ← getMatcherInfo? declName
      | return ReduceMatcherResult.notMatcher
    let args := e.getAppArgs
    let prefixSz := info.numParams + 1 + info.numDiscrs
    if args.size < prefixSz + info.numAlts then
      return ReduceMatcherResult.partialApp
    else
      let constInfo ← getConstInfo declName
      let f := constInfo.instantiateValueLevelParams declLevels
      let auxApp := mkAppN f args[0:prefixSz]
      let auxAppType ← inferType auxApp
      forallBoundedTelescope auxAppType info.numAlts fun hs _ => do
        let auxApp := mkAppN auxApp hs
        let auxApp ← whnf auxApp
        let auxAppFn := auxApp.getAppFn
        let mut i := prefixSz
        for h in hs do
          if auxAppFn == h then
            let result := mkAppN args[i] auxApp.getAppArgs
            let result := mkAppN result args[prefixSz + info.numAlts:args.size]
```

```
            return ReduceMatcherResult.reduced result.headBeta
          i := i + 1
        return ReduceMatcherResult.stuck auxApp
  | _ => pure ReduceMatcherResult.notMatcher

/- Given an expression `e`, compute its WHNF and if the result is a constructor, return field #i. -/
def project? (e : Expr) (i : Nat) : MetaM (Option Expr) := do
  let e ← whnf e
  matchConstCtor e.getAppFn (fun _ => pure none) fun ctorVal _ =>
    let numArgs := e.getAppNumArgs
    let idx := ctorVal.numParams + i
    if idx < numArgs then
      return some (e.getArg! idx)
    else
      return none

def reduceProj? (e : Expr) : MetaM (Option Expr) := do
  match e with
  | Expr.proj _ i c _ => project? c i
  | _                 => return none

/-
  Auxiliary method for reducing terms of the form `?m t_1 ... t_n` where `?m` is delayed assigned.
  Recall that we can only expand a delayed assignment when all holes/metavariables in the assigned value have been "filled".
-/
private def whnfDelayedAssigned? (f' : Expr) (e : Expr) : MetaM (Option Expr) := do
  if f'.isMVar then
    match (← getDelayedAssignment? f'.mvarId!) with
    | none => return none
    | some { fvars := fvars, val := val, .. } =>
      let args := e.getAppArgs
      if fvars.size > args.size then
        -- Insufficient number of argument to expand delayed assignment
        return none
      else
        let newVal ← instantiateMVars val
        if newVal.hasExprMVar then
           -- Delayed assignment still contains metavariables
           return none
        else
           let newVal := newVal.abstract fvars
           let result := newVal.instantiateRevRange 0 fvars.size args
           return mkAppRange result fvars.size args.size args
  else
    return none

/--
```

```
  Apply beta-reduction, zeta-reduction (i.e., unfold let local-decls), iota-reduction,
  expand let-expressions, expand assigned meta-variables. -/
partial def whnfCore (e : Expr) : MetaM Expr :=
  whnfEasyCases e fun e => do
    trace[Meta.whnf]! e
    match e with
    | Expr.const ..  => pure e
    | Expr.letE _ _ v b _ => whnfCore $ b.instantiate1 v
    | Expr.app f ..       =>
      let f := f.getAppFn
      let f' ← whnfCore f
      if f'.isLambda then
        let revArgs := e.getAppRevArgs
        whnfCore <| f'.betaRev revArgs
      else if let some eNew ← whnfDelayedAssigned? f' e then
        whnfCore eNew
      else
        let e := if f == f' then e else e.updateFn f'
        match (← reduceMatcher? e) with
        | ReduceMatcherResult.reduced eNew => whnfCore eNew
        | ReduceMatcherResult.partialApp   => pure e
        | ReduceMatcherResult.stuck _      => pure e
        | ReduceMatcherResult.notMatcher   =>
          matchConstAux f' (fun _ => return e) fun cinfo lvls =>
            match cinfo with
            | ConstantInfo.recInfo rec    => reduceRec rec lvls e.getAppArgs (fun _ => return e) whnfCore
            | ConstantInfo.quotInfo rec   => reduceQuotRec rec lvls e.getAppArgs (fun _ => return e) whnfCore
            | c@(ConstantInfo.defnInfo _) => do
              if (← isAuxDef c.name) then
                deltaBetaDefinition c lvls e.getAppRevArgs (fun _ => return e) whnfCore
              else
                return e
            | _ => return e
    | Expr.proj .. => match (← reduceProj? e) with
      | some e => whnfCore e
      | none => return e
    | _ => unreachable!

mutual
  /-- Reduce `e` until `idRhs` application is exposed or it gets stuck.
      This is a helper method for implementing smart unfolding. -/
  private partial def whnfUntilIdRhs (e : Expr) : MetaM Expr := do
    let e ← whnfCore e
    match (← getStuckMVar? e) with
    | some mvarId =>
      /- Try to "unstuck" by resolving pending TC problems -/
      if (← Meta.synthPending mvarId) then
```

```
      whnfUntilIdRhs e
    else
      return e -- failed because metavariable is blocking reduction
  | _ =>
    if isIdRhsApp e then
      return e -- done
    else
      match (← unfoldDefinition? e) with
      | some e => whnfUntilIdRhs e
      | none   => pure e -- failed because of symbolic argument

/--
  Auxiliary method for unfolding a class projection when transparency is set to `TransparencyMode.instances`.
  Recall that that class instance projections are not marked with `[reducible]` because we want them to be
  in "reducible canonical form".
-/
private partial def unfoldProjInst (e : Expr) : MetaM (Option Expr) := do
  if (← getTransparency) != TransparencyMode.instances then
    return none
  else
    match e.getAppFn with
    | Expr.const declName .. =>
      match (← getProjectionFnInfo? declName) with
      | some { fromClass := true, .. } =>
        match (← withDefault <| unfoldDefinition? e) with
        | none   => return none
        | some e =>
          match (← reduceProj? e.getAppFn) with
          | none   => return none
          | some r => return mkAppN r e.getAppArgs |>.headBeta
      | _ => return none
    | _ => return none

/-- Unfold definition using "smart unfolding" if possible. -/
partial def unfoldDefinition? (e : Expr) : MetaM (Option Expr) :=
  match e with
  | Expr.app f _ _ =>
    matchConstAux f.getAppFn (fun _ => unfoldProjInst e) fun fInfo fLvls => do
      if fInfo.levelParams.length != fLvls.length then
        return none
      else
        let unfoldDefault (_ : Unit) : MetaM (Option Expr) :=
          if fInfo.hasValue then
            deltaBetaDefinition fInfo fLvls e.getAppRevArgs (fun _ => pure none) (fun e => pure (some e))
          else
            return none
        if smartUnfolding.get (← getOptions) then
```

```
            let fAuxInfo? ← getConstNoEx? (mkSmartUnfoldingNameFor fInfo.name)
            match fAuxInfo? with
            | some fAuxInfo@(ConstantInfo.defnInfo _) =>
              deltaBetaDefinition fAuxInfo fLvls e.getAppRevArgs (fun _ => pure none) fun e₁ => do
                let e₂ ← whnfUntilIdRhs e₁
                if isIdRhsApp e₂ then
                  return some (extractIdRhs e₂)
                else
                  return none
            | _ => unfoldDefault ()
          else
            unfoldDefault ()
    | Expr.const declName lvls _ => do
      if smartUnfolding.get (← getOptions) && (← getEnv).contains (mkSmartUnfoldingNameFor declName) then
        return none
      else
        let (some (cinfo@(ConstantInfo.defnInfo _))) ← getConstNoEx? declName | pure none
        deltaDefinition cinfo lvls
          (fun _ => pure none)
          (fun e => pure (some e))
    | _ => return none
end

def unfoldDefinition (e : Expr) : MetaM Expr := do
  let some e ← unfoldDefinition? e | throwError! "failed to unfold definition{indentExpr e}"
  return e

@[specialize] partial def whnfHeadPred (e : Expr) (pred : Expr → MetaM Bool) : MetaM Expr :=
  whnfEasyCases e fun e => do
    let e ← whnfCore e
    if (← pred e) then
        match (← unfoldDefinition? e) with
        | some e => whnfHeadPred e pred
        | none   => return e
    else
      return e

def whnfUntil (e : Expr) (declName : Name) : MetaM (Option Expr) := do
  let e ← whnfHeadPred e (fun e => return !e.isAppOf declName)
  if e.isAppOf declName then
    return e
  else
    return none

/-- Try to reduce matcher/recursor/quot applications. We say they are all "morally" recursor applications. -/
def reduceRecMatcher? (e : Expr) : MetaM (Option Expr) := do
  if !e.isApp then
```

```
    return none
  else match (← reduceMatcher? e) with
    | ReduceMatcherResult.reduced e => return e
    | _ => matchConstAux e.getAppFn (fun _ => pure none) fun cinfo lvls => do
      match cinfo with
      | ConstantInfo.recInfo «rec»  => reduceRec «rec» lvls e.getAppArgs (fun _ => pure none) (fun e => pure (some e))
      | ConstantInfo.quotInfo «rec» => reduceQuotRec «rec» lvls e.getAppArgs (fun _ => pure none) (fun e => pure (some e))
      | c@(ConstantInfo.defnInfo _) =>
        if (← isAuxDef c.name) then
          deltaBetaDefinition c lvls e.getAppRevArgs (fun _ => pure none) (fun e => pure (some e))
        else
          return none
      | _ => return none

unsafe def reduceBoolNativeUnsafe (constName : Name) : MetaM Bool := evalConstCheck Bool `Bool constName
unsafe def reduceNatNativeUnsafe (constName : Name) : MetaM Nat := evalConstCheck Nat `Nat constName
@[implementedBy reduceBoolNativeUnsafe] constant reduceBoolNative (constName : Name) : MetaM Bool
@[implementedBy reduceNatNativeUnsafe] constant reduceNatNative (constName : Name) : MetaM Nat

def reduceNative? (e : Expr) : MetaM (Option Expr) :=
  match e with
  | Expr.app (Expr.const fName _ _) (Expr.const argName _ _) _ =>
    if fName == `Lean.reduceBool then do
      return toExpr (← reduceBoolNative argName)
    else if fName == `Lean.reduceNat then do
      return toExpr (← reduceNatNative argName)
    else
      return none
  | _ =>
    return none

@[inline] def withNatValue {α} (a : Expr) (k : Nat → MetaM (Option α)) : MetaM (Option α) := do
  let a ← whnf a
  match a with
  | Expr.const `Nat.zero _ _      => k 0
  | Expr.lit (Literal.natVal v) _ => k v
  | _                             => return none

def reduceUnaryNatOp (f : Nat → Nat) (a : Expr) : MetaM (Option Expr) :=
  withNatValue a fun a =>
  return mkNatLit <| f a

def reduceBinNatOp (f : Nat → Nat → Nat) (a b : Expr) : MetaM (Option Expr) :=
  withNatValue a fun a =>
  withNatValue b fun b => do
  trace[Meta.isDefEq.whnf.reduceBinOp]! "{a} op {b}"
  return mkNatLit <| f a b
```

```
def reduceBinNatPred (f : Nat → Nat → Bool) (a b : Expr) : MetaM (Option Expr) := do
  withNatValue a fun a =>
  withNatValue b fun b =>
  return toExpr <| f a b

def reduceNat? (e : Expr) : MetaM (Option Expr) :=
  if e.hasFVar || e.hasMVar then
    return none
  else match e with
    | Expr.app (Expr.const fn _ _) a _                  =>
      if fn == `Nat.succ then
        reduceUnaryNatOp Nat.succ a
      else
        return none
    | Expr.app (Expr.app (Expr.const fn _ _) a1 _) a2 _ =>
      if fn == `Nat.add then reduceBinNatOp Nat.add a1 a2
      else if fn == `Nat.sub then reduceBinNatOp Nat.sub a1 a2
      else if fn == `Nat.mul then reduceBinNatOp Nat.mul a1 a2
      else if fn == `Nat.div then reduceBinNatOp Nat.div a1 a2
      else if fn == `Nat.mod then reduceBinNatOp Nat.mod a1 a2
      else if fn == `Nat.beq then reduceBinNatPred Nat.beq a1 a2
      else if fn == `Nat.ble then reduceBinNatPred Nat.ble a1 a2
      else return none
    | _ =>
      return none


@[inline] private def useWHNFCache (e : Expr) : MetaM Bool := do
  -- We cache only closed terms without expr metavars.
  -- Potential refinement: cache if `e` is not stuck at a metavariable
  if e.hasFVar || e.hasExprMVar then
    return false
  else
    match (← getConfig).transparency with
    | TransparencyMode.default => true
    | TransparencyMode.all     => true
    | _                        => false

@[inline] private def cached? (useCache : Bool) (e : Expr) : MetaM (Option Expr) := do
  if useCache then
    match (← getConfig).transparency with
    | TransparencyMode.default => return (← get).cache.whnfDefault.find? e
    | TransparencyMode.all     => return (← get).cache.whnfAll.find? e
    | _                        => unreachable!
  else
    return none
```

```
private def cache (useCache : Bool) (e r : Expr) : MetaM Expr := do
  if useCache then
    match (← getConfig).transparency with
    | TransparencyMode.default => modify fun s => { s with cache.whnfDefault := s.cache.whnfDefault.insert e r }
    | TransparencyMode.all     => modify fun s => { s with cache.whnfAll     := s.cache.whnfAll.insert e r }
    | _                        => unreachable!
  return r

partial def whnfImp (e : Expr) : MetaM Expr :=
  whnfEasyCases e fun e => do
    checkMaxHeartbeats "whnf"
    let useCache ← useWHNFCache e
    match (← cached? useCache e) with
    | some e' => pure e'
    | none    =>
      let e' ← whnfCore e
      match (← reduceNat? e') with
      | some v => cache useCache e v
      | none   =>
        match (← reduceNative? e') with
        | some v => cache useCache e v
        | none   =>
          match (← unfoldDefinition? e') with
          | some e => whnfImp e
          | none   => cache useCache e e'

@[builtinInit] def setWHNFRef : IO Unit :=
  whnfRef.set whnfImp

builtin_initialize
  registerTraceClass `Meta.whnf

end Lean.Meta
```