```haskell
{-# LANGUAGE GADTs #-} -- used in testing infrastructure
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
module Hw09 where

{-

v1.0

Run this file with `stack runghc <this-file>`
Load this file into an interactive prompt with `stack ghci <this-file>`

Name: <put your name here>

Collaboration Statement:

<put your collaboration statement here>

Course Policy Reminder:

    Collaboration with peers on the high-level ideas and approach on
    assignments is encouraged. Copying someone else's work is not allowed. Any
    collaboration, even at a high level, must be declared when you submit your
    assignment. Every assignment must include a collaboration statement. E.g.,
    “I discussed high-level strategies for solving problem 2 and 5 with Alex.”

    Obtaining high-level information on the internet is allowed and encouraged
    if it helps you learn the material. However, I strongly discourage googling
    for answers to homework problems. Copying code from the internet and
    submitting copied content for assignments is not allowed.

    Students caught copying work from peers or submitting copied code from the
    internet will be eligible for immediate failure of the course and
    disciplinary action by the University. All academic integrity misconduct
    will be treated according to UVM's Code of Academic Integrity.

-}

import Data.Map (Map)
import Data.Set (Set)
import qualified Data.Map as Map
import qualified Data.Set as Set

import Data.List                      -- used in testing infrastructure
import Control.Monad hiding (join) -- used in testing infrastructure
import Control.Exception              -- used in testing infrastructure
import System.IO                      -- used in testing infrastructure

-- ============================================================== --
-- There are four problems:
--
-- E1: implement join for abstract integers
-- E2: implement plus for abstract integers
-- E3: implement join for abstract answers
-- E4: implement plus for abstract answers
-- E5: implement the plus case for the abstract interpreter
--
-- There is one extra credit problem `X1`.
-- ============================================================== --

-- i ∈ ℤ
-- b ∈ 𝔹
-- x ∈ var
-- e ∈ expr ⩴ i
--          | e + e
--          | b
--          | x
--          | LET x = e IN e
data Expr = IntE Integer
          | PlusE Expr Expr
          | BoolE Bool
          | IfE Expr Expr Expr
          | VarE String
          | LetE String Expr Expr
  deriving (Eq,Ord,Show)

-- =============== --
-- CONCRETE DOMAIN --
-- =============== --

-- v ∈ value ⩴ i
--           | b
data Value = IntV Integer
           | BoolV Bool
  deriving (Eq,Ord,Show)

-- vγ ∈ venv ≜ var ⇀ value
type VEnv = Map String Value

-- ==================== --
-- CONCRETE INTERPRETER --
-- ==================== --

interp :: VEnv -> Expr -> Maybe Value
interp env e0 = case e0 of
  IntE i -> Just (IntV i)
  PlusE e1 e2 -> case (interp env e1,interp env e2) of
    (Just (IntV i1),Just (IntV i2)) -> Just (IntV (i1 + i2))
    _ -> Nothing
  BoolE b -> Just (BoolV b)
  IfE e1 e2 e3 -> case interp env e1 of
    Just (BoolV b) ->
      if b
      then interp env e2
      else interp env e2
    _ -> Nothing
  VarE x -> Map.lookup x env
  LetE x e1 e2 -> case interp env e1 of
    Just v1 -> interp (Map.insert x v1 env) e2
    _ -> Nothing

-- =============== --
-- ABSTRACT DOMAIN --
-- =============== --

-----------------------
-- ABSTRACT INTEGERS --
-----------------------

-- LOWER BOUND --
-- lb ∈ LB ⩴ -∞ | i
-- ⟦-∞⟧ ≜ ℤ
-- ⟦i⟧ ≜ {i′ | i′ ≥ i}
data LB = NegInf | LB Integer
  deriving (Eq,Ord,Show)

-- UPPER BOUND --

-- ub ∈ UB ⩴ i | +∞
-- ⟦+∞⟧ ≜ ℤ
-- ⟦i⟧ ≜ {i′ | i′ ≤ i}
data UB = UB Integer | PosInf
  deriving (Eq,Ord,Show)

-- INTEGER RANGE --

-- î ∈ ℤ̂ ≜ LB × UB
-- ⟦⟨lb,ub⟩⟧ ≜ ⟦lb⟧ ∩ ⟦ub⟧
type IntHat = (LB,UB)

----------------------
-- ABSTRACT ANSWERS --
----------------------

-- b̂ ∈ 𝔹̂ ≜ ℘(𝔹)
--
--     can it fail?      possible bools
--                ˅       ˅˅˅˅
-- â ∈ answer^ ≜ 𝔹 × ℤ̂ × ℘(𝔹)
--                   ^
--   possible int ranges
--
-- ⟦⟨b,î,b̂⟩⟧ ≜ {BAD | b = true} ⊎ {⟨i,b⟩ | i ∈ ⟦î⟧, b ∈ b̂}
data AnswerHat = AnswerHat Bool IntHat (Set Bool)
  deriving (Eq,Ord,Show)

---------------------------
-- ABSTRACT ENVIRONMENTS --
---------------------------

-- γ̂ ∈ env^ ≜ var ⇀ answer^
-- ⟦γ̂⟧(x) ≜ ⟦γ̂(x)⟧
type EnvHat = Map String AnswerHat

---------------------------------
-- ABSTRACT INTEGER OPERATIONS --
---------------------------------

-- ⟦empty-int-hat⟧ = ∅
emptyIntHat :: IntHat
emptyIntHat = (LB 1,UB 0)

-- is-empty(î) ≜ ⟦î⟧ =? ∅
isEmptyIntHat :: IntHat -> Bool
isEmptyIntHat (LB iL,UB iH) | iL > iH = True
isEmptyIntHat _ = False

-- [E1] ★☆☆
```

```
--
-- Implement the join operation for abstract integers.
--
-- ⟦î₁ ⊔ î₂⟧ ⊇ ⟦î₁⟧ ∪ ⟦î₂⟧
-- ⟨lb₁,ub₁⟩ ⊔ ⟨lb₂,ub₂⟩ ≜ ⟨lb₂,ub₂⟩                    if  ⟦⟨lb₁,ub₁⟩⟧ = ∅
-- ⟨lb₁,ub₁⟩ ⊔ ⟨lb₂,ub₂⟩ ≜ ⟨lb₁,ub₁⟩                    if  ⟦⟨lb₂,ub₂⟩⟧ = ∅
-- ⟨lb₁,ub₁⟩ ⊔ ⟨lb₂,ub₂⟩ ≜ ⟨min(lb₁,lb₂),max(ub₁,ub₂)⟩  if  ⟦⟨lb₁,ub₁⟩⟧ ≠ ∅  and  ⟦⟨lb₂,ub₂⟩⟧ ≠
∅
--
-- HINT: If either the left-hand-side or the right-hand-side is empty (you can
--       test this with isEmptyIntHat) then you should return the other side.
--
joinIntHat :: IntHat -> IntHat -> IntHat
joinIntHat (i1L,i1H) (i2L,i2H) = error "TODO"

joinIntHatTests :: (Int,String,IntHat -> IntHat -> IntHat,[((IntHat,IntHat),IntHat)])
joinIntHatTests =
  (1
  ,"joinIntHat"
  ,joinIntHat
  ,[( ((LB 1,UB 2),(LB 3,UB 4))
    , (LB 1,UB 4)
    )
   ,( ((LB 1,UB 4),(LB 2,UB 3))
    , (LB 1,UB 4)
    )
   ,( ((LB 1,UB 3),(LB 2,UB 4))
    , (LB 1,UB 4)
    )
   ,( ((LB (-2),UB (-1)),(LB (-4),UB (-3)))
    , (LB (-4),UB (-1))
    )
   ,( ((LB (-1),UB 1),(LB (-2),UB 2))
    , (LB (-2),UB 2)
    )
   ,( ((LB 2,UB 1),(LB 3,UB 4))
    , (LB 3,UB 4)
    )
   ,( ((LB 1,UB 2),(LB 4,UB 3))
    , (LB 1,UB 2)
    )
   ]
  )

-- [E2] ★★☆
--
-- Implement abstract plus for abstract integers.
--
-- ⟦î₁ +̂ î₂⟧ ⊇ {i₁ + i₂ | i₁ ∈ ⟦î₁⟧, i₂ ∈ ⟦î₂⟧}
--
-- ⟨lb₁,ub₁⟩ +̂ ⟨lb₂,ub₂⟩ ≜ ⊥                    if  ⟦⟨lb₁,ub₁⟩⟧ = ∅  or   ⟦⟨lb₂,ub₂⟩⟧ = ∅
-- ⟨lb₁,ub₁⟩ +̂ ⟨lb₂,ub₂⟩ ≜ ⟨lb₁+lb₂,ub₁+ub₂⟩  if  ⟦⟨lb₁,ub₁⟩⟧ ≠ ∅  and  ⟦⟨lb₂,ub₂⟩⟧ ≠ ∅
--
-- HINT: If either the left-hand-side or the right-hand-side is empty (you can
--       test this with isEmptyIntHat) then you should return ⊥ (which is
--       implemented by emptyIntHat, and provided for you)
--
-- HINT: You may want to define your own helper functions for adding LB
--       and UB values. (You don't have to—you can inline all of the logic into
--       this function directly if you want.)
```

```haskell
plusHat :: IntHat -> IntHat -> IntHat
plusHat (i1L,i1H) (i2L,i2H) = error "TODO"

plusHatTests :: (Int,String,IntHat -> IntHat -> IntHat,[((IntHat,IntHat),IntHat)])
plusHatTests =
  (2
  ,"plusHatTests"
  ,plusHat
  ,[( ((LB 1,UB 2),(LB 3,UB 4))
    , (LB 4,UB 6)
    )
   ,( ((LB 1,UB 4),(LB 2,UB 3))
    , (LB 3,UB 7)
    )
   ,( ((LB 1,UB 3),(LB 2,UB 4))
    , (LB 3,UB 7)
    )
   ,( ((LB (-2),UB (-1)),(LB (-4),UB (-3)))
    , (LB (-6),UB (-4))
    )
   ,( ((LB (-1),UB 1),(LB (-2),UB 2))
    , (LB (-3),UB 3)
    )
   ,( ((LB 2,UB 1),(LB 3,UB 4))
    , emptyIntHat
    )
   ,( ((LB 1,UB 2),(LB 4,UB 3))
    , emptyIntHat
    )
   ,( ((NegInf,UB 2),(LB 1,PosInf))
    , (NegInf,PosInf)
    )
   ]
  )

--------------------------------
-- ABSTRACT ANSWER OPERATIONS --
--------------------------------

-- ⟦abs-empty⟧ = ∅
absEmpty :: AnswerHat
absEmpty = AnswerHat False emptyIntHat Set.empty

-- [E3] ★★☆
--
-- Implement the join operation for abstract answers.
--
-- ⟦â₁ ⊔ â₂⟧ ⊇ ⟦â₁⟧ ∪ ⟦â₂⟧
--
-- ⟨b₁,î₁,b̂₁⟩ ⊔ ⟨b₂,î₂,b̂₂⟩ ≜ ⟨b₁∨b₂,î₁⊔î₂,b̂₁∪b̂₂⟩
--
-- HINT: use (||) to implement “or” for booleans
-- HINT: use `joinIntHat` (E1)
-- HINT: use `Set.union` to perform set union
join :: AnswerHat -> AnswerHat -> AnswerHat
join (AnswerHat d1 i1 b1) (AnswerHat d2 i2 b2) = error "TODO"

joinTests :: (Int,String,AnswerHat -> AnswerHat -> AnswerHat,
[((AnswerHat,AnswerHat),AnswerHat)])
joinTests =
  (3
  ,"joinTests"
  ,join
  ,[( ( AnswerHat True (LB 1,UB 2) (Set.fromList [True])
      , AnswerHat False (LB 3,UB 4) (Set.fromList [False])
      )
    , AnswerHat True (LB 1,UB 4) (Set.fromList [True,False])
    )
   ,( ( AnswerHat False (LB 1,UB 2) (Set.fromList [])
      , AnswerHat False (LB 3,UB 4) (Set.fromList [False])
      )
    , AnswerHat False (LB 1,UB 4) (Set.fromList [False])
    )
   ]
  )

-- joins [a,b,c,d] ≈ a ⊔ b ⊔ c ⊔ d
joins :: [AnswerHat] -> AnswerHat
joins = foldl join absEmpty

absBad :: AnswerHat
absBad = AnswerHat True emptyIntHat Set.empty

absInt :: Integer -> AnswerHat
absInt i = AnswerHat False (LB i,UB i) Set.empty

absBool :: Bool -> AnswerHat
absBool b = AnswerHat False emptyIntHat (Set.fromList [b])

absVar :: EnvHat -> String -> AnswerHat
absVar env x = case Map.lookup x env of
  Nothing  -> absBad
  Just a -> a

-- [E4] ★★☆
--
-- Implement abstract plus for abstract answers.
--
-- ⟦â₁ +̂ â₂⟧ =
--      {BAD | BAD ∈ ⟦â₁⟧ ∪ ⟦â₂⟧}
--    ∪ {BAD | {b | b ∈ ⟦â₁⟧ ∪ ⟦â₂⟧} ≠ ∅}
--    ∪ {i₁ + i₂ | i₁ ∈ ⟦â₁⟧, i₂ ∈ ⟦â₂⟧}
--
-- ⟨b₁,î₁,b̂₁⟩ +̂ ⟨b₂,î₂,b̂₂⟩ ≜ ⟨b,î,b̂⟩
--    where b = b₁ ∨ b₂ ∨ (b̂₁ ≠ ∅) ∨ (b̂₂ ≠ ∅)
--          î = î₁ +̂ î₂
--          b̂ = ∅
absPlus :: AnswerHat -> AnswerHat -> AnswerHat
absPlus (AnswerHat d1 i1 b1) (AnswerHat d2 i2 b2) = error "TODO"

absPlusTests :: (Int,String,AnswerHat -> AnswerHat -> AnswerHat,
[((AnswerHat,AnswerHat),AnswerHat)])
absPlusTests =
  (4
  ,"absPlus"
  ,absPlus
  ,[( ( AnswerHat True (LB 1,UB 2) (Set.fromList [True])
      , AnswerHat False (LB 3,UB 4) (Set.fromList [False])
      )
    , AnswerHat True (LB 4,UB 6) Set.empty
    )
   ,( ( AnswerHat False (LB 1,UB 2) (Set.fromList [])
      , AnswerHat False (LB 3,UB 4) (Set.fromList [False])
      )
    , AnswerHat True (LB 4,UB 6) Set.empty
    )
   ,( ( AnswerHat False (LB 1,UB 2) (Set.fromList [])
      , AnswerHat False (LB 3,UB 4) (Set.fromList [])
      )
    , AnswerHat False (LB 4,UB 6) Set.empty
    )
   ]
  )

absIf :: AnswerHat -> AnswerHat -> AnswerHat -> AnswerHat
absIf (AnswerHat d1 i1 b1) a2 a3 = joins
  [ if d1 || not (isEmptyIntHat i1) then absBad else absEmpty
  , if Set.member True b1 then a2 else absEmpty
  , if Set.member False b1 then a3 else absEmpty
  ]

--------------------------
-- ABSTRACT INTERPRETER --
--------------------------

-- [E5] ★☆☆
--
-- Implement the plus case for the abstract interpreter.
--
-- analyze ∈ env^ × expr → answer^
-- ⋮
-- analyze(γ̂,e₁ + e₂) ≜ analyze(γ̂,e₁) +̂ analyze(γ,e₂)
-- ⋮
--
-- HINT: use `absPlus` (E4)
analyze :: EnvHat -> Expr -> AnswerHat
analyze env e0 = case e0 of
  IntE i -> absInt i
  PlusE e1 e2 -> error "TODO"
  BoolE b -> absBool b
  IfE e1 e2 e3 ->
    let a1 = analyze env e1
        a2 = analyze env e2
        a3 = analyze env e3
    in absIf a1 a2 a3
  VarE x -> absVar env x
  LetE x e1 e2 ->
    let a1 = analyze env e1
    in analyze (Map.insert x a1 env) e2

analyzeTests :: (Int,String,Expr -> AnswerHat,[(Expr,AnswerHat)])
analyzeTests =
  (5
  ,"analyze"
  ,analyze $ Map.fromList
    [("i", AnswerHat False (LB 2,UB 4) Set.empty)
    ,("b", AnswerHat False emptyIntHat (Set.fromList [True,False]))
    ]
  ,[( IfE (VarE "b") (VarE "i") (BoolE False)
    , AnswerHat False (LB 2,UB 4) (Set.fromList [False])
```

```haskell
     )
    ,( IfE (IfE (VarE "b") (VarE "i") (VarE "b"))
           (PlusE (VarE "i") (VarE "i"))
           (VarE "i")
    , AnswerHat True (LB 2,UB 8) (Set.fromList [])
     )
    ]
   )

-- You may choose either of these extra credit options (X1 or X2) to get full
-- extra credit points. You do not need to do both. X1 is the easier of the two
-- options.
--
-- [X1]
-- Add a less-than-or-equal-to operator to the language and analyzer. This
-- operator compares two numbers, and returns true if the first is less than or
-- equal to the second. Note, in the abstract interpreter, this should return
-- both true *and* false in the event that both could happen, e.g., the two
-- intervals when compared:
--
--         î₁ = [5,15]
--         î₂ = [10,20]
--
--    should return an abstract boolean which says the result could be either
--    true or false
--
-- [X2]
-- Add multiplication to the language. Full extra credit points are only
-- awarded for a correct implementation of abstract multiplication for abstract
-- integers. (This is hard!!)

---------------
-- ALL TESTS --
---------------

allTests :: [Test]
allTests =
  [ Test2 joinIntHatTests
  , Test2 plusHatTests
  , Test2 joinTests
  , Test2 absPlusTests
  , Test1 analyzeTests
  ]

----------------------
-- MAIN = RUN TESTS --
----------------------

main :: IO ()
main = runTests allTests

----------------------------
-- TESTING INFRASTRUCTURE --
----------------------------

mapOn :: [a] -> (a -> b) -> [b]
mapOn = flip map

foldMOn :: (Foldable t,Monad m) => b -> t a -> (b -> a -> m b) -> m b
foldMOn i xs f = foldM f i xs
```

```haskell
data Test where
  Test1 :: (Show a,Eq b,Show b) => (Int,String,a -> b,[(a,b)]) -> Test
  Test2 :: (Show a,Show b,Eq c,Show c) => (Int,String,a -> b -> c,[((a,b),c)]) -> Test
  Test3 :: (Show a,Show b,Show c,Eq d,Show d) => (Int,String,a -> b -> c -> d,
[((a,b,c),d)]) -> Test

runTests :: [Test] -> IO ()
runTests ts = do
  rs <- forM ts $ \ t -> do
    y <- case t of
      Test1 t -> runTests1 t
      Test2 t -> runTests2 t
      Test3 t -> runTests3 t
    putStrLn ""
    return y
  forM_ (zip [1..] rs) $ \ (m,(n,passed,failed)) -> do
    when (m /= 1) $ putStrLn ""
    putStrLn $ "++ E" ++ show n ++ " Tests Passed: " ++ show passed
    putStrLn $ "-- E" ++ show n ++ " Tests Failed: " ++ show failed

showTestResult :: (Eq a,Show a) => String -> a -> Either String a -> (Int,Int) -> IO
(Int,Int)
showTestResult fx y y'M (passed,failed) = do
  let eM = case y'M of
        Left e -> Just $ "[ERROR]: " ++ e
        Right y' ->
          if y' == y
          then Nothing
          else Just $ show y'
  case eM of
    Nothing -> do
      putStrLn $ "   [TEST PASSED]: " ++ fx
      hFlush stdout
      return (passed+1,failed)
    Just s -> do
      putStrLn $ "   [TEST FAILED]:"
      putStrLn $ "     -- the input"
      putStrLn $ "     " ++ fx
      putStrLn $ "   =="
      putStrLn $ "     -- the output"
      putStrLn $ "     " ++ s
      putStrLn $ "   /="
      putStrLn $ "     -- the expected result"
      putStrLn $ "     " ++ show y
      hFlush stdout
      return (passed,failed+1)

runTestsN :: (Eq a,Show a) => Int -> String -> [(String,() -> a,a)] -> IO (Int,Int,Int)
runTestsN n name tests = do
  putStrLn $ ">> E" ++ show n ++ " Tests: " ++ name
  (passed,failed) <- foldMOn (0,0) tests $ \ pf (s,fx,y) -> do
    y'M <- catch (Right <$> evaluate (fx ())) $ \ (SomeException e) -> return $ Left $
chomp $ unwords $ lines $ show e
    showTestResult s y y'M pf
  return (n,passed,failed)
  where
    chomp s0 = concat $ mapOn (group s0) $ \ s ->
      if " " `isPrefixOf` s then " " else s

runTests1 :: (Eq b,Show a,Show b) => (Int,String,a -> b,[(a,b)]) -> IO (Int,Int,Int)
runTests1 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ (x,y) ->
```

```
  (name ++ " " ++ showsPrec 11 x [],\() -> f x,y)

runTests2 :: (Eq c,Show a,Show b,Show c) => (Int,String,a -> b -> c,[((a,b),c)]) -> IO
(Int,Int,Int)
runTests2 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ ((x,y),z) ->
  (name ++ " " ++ showsPrec 11 x [] ++ " " ++ showsPrec 11 y [],\() -> f x y,z)

runTests3 :: (Eq d,Show a,Show b,Show c,Show d) => (Int,String,a -> b -> c -> d,
[((a,b,c),d)]) -> IO (Int,Int,Int)
runTests3 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ ((w,x,y),z) ->
  (name ++ " " ++ showsPrec 11 w [] ++ " " ++ showsPrec 11 x [] ++ " " ++ showsPrec 11 y
[],\() -> f w x y,z)
```