My Report

Me

Tuesday 18th March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LATEXeditor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.

- To open ghci and play with your code: `stack ghci`

- To run the executable from Section **??**: `stack build && stack exec myprogram`

- To run the tests from Section **??**: `stack clean && stack test --coverage`

2 The library files

3 The AC3Solver library

This section describes a module which we will import later on.

```
module AC3Solver where

import Control.Monad.Writer
    ( runWriter, MonadWriter(tell), Writer )
```

To start of, we define the AC3 instance. For each agent, we have a set of agents of type `a`. An AC3 instance then constains a list of constraints `constraintAA`, and a list of domains. Each `constraintAA` contains a pair of agents (X,Y), and then a function, such as `(==)`, which is the constraint on the arc from X to Y. Each `Domain` item contains an agent, and then a list of values of type `b`.

We may have multiple constraints for a pair of agentsd (X,Y), such as both `(>)` and `(>=)`. The programme however expects that each agent has exactly 1 domain specified for it, and each constraint (X,Y) should have some (possibly empty) domain assigned to it.

```
data AC3 a b = AC3 {
    -- Constraint should take values from the first & second agents as params x & y resp.
       in \x,y-> x ?=? y.
    -- We should allow for multiple constraints for (X,Y), eg. both (x > y) AND (x < y) in
       the set.
    cons :: [ConstraintAA a b],
    -- Assume we have 1 domain list for each variable. (TODO: Check for this? )
    domains :: [Domain a b] }

type Domain a b = (Agent a, [b])
type ConstraintAA a b = (Agent a, Agent a, Constraint b)
type Constraint a = a -> a -> Bool

type Agent a = a
```

For each constraint (X,Y,f), we want to check for each value in the domain of X whether there is a value in Y's domain such that `f x y` is satisfied. Values of X for which there is no such

value in Y are removed from X's domain. We make use of the Writer monad to do an $O(1)$ lookup to see if we removed items from X's domain.

```
-- Return the elements of xs for which there exist a y \in ys, such that c x y holds.
-- Using the writer monad, we also give a O(1) method to check whether we altered x's
    domain after termination.
checkDomain :: [a] -> [a] -> Constraint a -> Writer String [a]
checkDomain [] _ _ = return []
checkDomain (x:xs) ys c = do
    rest <- checkDomain xs ys c
    if not $ null [ y' | y'<-ys, c x y'] then return $ x:rest
    else tell "Altered domain" >> return rest -- This is nicely formatted for readability,
         it could just be "." or whatever.
```

Each time we call iterate, we start of by looking for the domains of agents X Y for our constraint (X,Y). Once we find these, we are likely to replace the original domain for X with a reduced one. We use `popXy` and `popX` to find the domains for X Y, and at the same time we also remove the *old* domain for X. Using this is 1 walk through the list, and saves us 2 walks. (Separate lookup for y, and a walk to delete the old x.) Once we have checked the current constraint, we then add back the *new* domain for X.

```
-- PRE: x is an element of (a:as)
popX :: Eq a => Agent a -> [Domain a b] -> ([b], [Domain a b] )
popX _ [] = undefined -- should not occur.
popX x (a@(aA, aD):as) = if x == aA then (aD,as) else let (x', as') = popX x as in (x', a:
    as')

-- PRE: x != y; x,y are elements of (a:as).
--      (else, this is not a binary constraint but a unary one.)
popXy :: Eq a => Agent a -> Agent a -> [Domain a b] -> ([b], [b], [Domain a b] )
popXy _ _ [] = undefined -- should not occur.
popXy x y (a@(aA, aD):as)
    | x == aA = let -- we want to REMOVE a from the list.
        -- search through the rest of the list and return y's domain.
        yDomain = head [b' | (a',b')<-as, y==a' ]
        in (aD, yDomain, as)
    | y == aA = let (retX, retAs) = popX x as in (retX, aD, a:retAs)
    | otherwise = let (retX, retY, retAs) = popXy x y as in (retX, retY, a:retAs)
```

We now come to the main part of the algorithm. The iterateAC3 function runs as long as the queue of constraints is not empty, starting with the original set of constraints. We get the domains of X Y, and remove the *old* domains of X. We then run checkDomain, and add the new domain of X back to the list of domains. If X's domain was altered, then we add all constraints of the form (Y,X) to the back of the queue.

```
ac3 :: (Ord a, Ord b) => AC3 a b -> [Domain a b] -- return a list of domains.
ac3 m@(AC3 c d) = let
    queue = c -- put each constraint into the queue.  -- TODO: implement this better, eg a
        priority queue?
    in iterateAC3 m queue d

iterateAC3 :: (Ord a, Ord b) => AC3 a b -> [ConstraintAA a b] -> [Domain a b] -> [Domain a
    b]
iterateAC3 _ [] d = d
iterateAC3 m@(AC3 fullCS _) ((x,y,c):cs) d = let
    (xDomain, yDomain, alteredD) = popXy x y d
    (newX, str) = runWriter $ checkDomain xDomain yDomain c
        -- In a lens, we could do this with "modify (\ (a,_) -> (a, newX))"
    newDomains = (x, newX) : alteredD
    z = if null str then cs else cs ++ [c' | c'@(y1,x1,_)<-fullCS, y1==y, x1==x ] -- take
        all constraints of the form (y,x, c)
    in iterateAC3 m z newDomains
```

```
module NQueens where
```

3

```haskell
import AC3Solver ( ac3, AC3(AC3) )  -- Import AC3 solver
import Backtracking (findSolution, findAllSolutions ) -- Import backtracking solver

notSameQueenMove :: (Int, Int) -> (Int, Int) -> Bool
notSameQueenMove (a1, a2) (b1, b2) =
    not (a1 == b1 || a2 == b2 || abs (a1 - b1) == abs (a2 - b2))

(//=) :: (Int, Int) -> (Int, Int) -> Bool
(a1, a2) //= (b1, b2) = notSameQueenMove (a1, a2) (b1, b2)

nQueens :: Int -> AC3 Int (Int, Int)
nQueens n = let
    agents = [0 .. n-1] -- queens as row numbers
    domain = [(row, [(row, col) | col <- [0 .. n-1]]) | row <- agents] --1 queen per row
    constraints = [(a, b, (//=)) | a <- agents, b <- agents, a < b]
    in AC3 constraints domain

-- uses old backtracking -- a single solution (unused)
solveNQueens :: Int -> Maybe [(Int, (Int, Int))]
solveNQueens n = findSolution (AC3 constraints (ac3 (nQueens n)))
  where
    AC3 constraints _ = nQueens n

-- Find all solutions
solveAllNQueens :: Int -> [[(Int, (Int, Int))]]
solveAllNQueens n = findAllSolutions (AC3 constraints (ac3 (nQueens n)))
  where
    AC3 constraints _ = nQueens n

-- simple pretty print using number as queens and . as empty spaces
prettyPrintBoard :: Int -> [(Int, (Int, Int))] -> IO ()
prettyPrintBoard n solution = do
    let board = [[if (r, c) `elem` map snd solution then show r else "." | c <- [0 .. n-1]]
          | r <- [0 .. n-1]]
    mapM_ (putStrLn . pptHelper) board
    putStrLn ""

pptHelper :: [String] -> String
pptHelper [] = ""
pptHelper [x] = x
pptHelper (x:xs) = x ++ " " ++ pptHelper xs

nQueensMain :: IO ()
nQueensMain = do
    putStrLn "Enter board size (N):"
    n <- readLn
    let solutions = solveAllNQueens n
    --let solutions = solveNQueens n --uncomment for 1 solution instead
    if null solutions
    then putStrLn "No solution found."
    else do
        putStrLn "Solutions: "
        mapM_ (prettyPrintBoard n) solutions
        putStrLn $ "Found " ++ show (length solutions) ++ " solution(s)"
```

```haskell
--{-# LANGUAGE LambdaCase #-} -- todo remove? if not using data.graph.read...
module GraphCol where

import Control.Monad (when)
import Data.Char (toUpper)
import Data.Graph
--import Data.Graph.Read
import Data.List
import Text.Read (readMaybe)
import Test.QuickCheck

import AC3Solver
import Backtracking
import Scheduling (parseInput)

-- We define a newtype, so that we can generate arbitrary instances.
newtype GraphCol = GC (AC3 Vertex Int)
```

```
seqPair :: (Gen a, Gen a) -> Gen (a,a)
seqPair (ma, mb) = ma >>= \a -> mb >>= \b -> return (a,b)

-- (Seems graphs don't already have an arbitrary instance...)
instance Arbitrary GraphCol where
    arbitrary = sized arbitGraphColN where
        arbitGraphColN n = do
            nColours <- choose (0, n 'div' 2)
            sizeV <- choose (0, n) -- we make vertices 0..sizeV INCLUDING SIZEV!!
            sizeE <- choose (0,n)
            e <- sequence [seqPair (choose (0, sizeV), choose (0, sizeV)) | _<-[0..sizeE]]
            let g = buildG (0, sizeV) e
            return $ convertGraphToAC3 g nColours --return $ convertGraphToAC3 g n

instance Show GraphCol where
  --show :: GraphCol -> String
  show (GC (AC3 c d)) = let
    strCon = "[" ++ makeShow c ++ "]" where
      makeShow [] = ""
      makeShow ((x,y,_):cs) =
        "(" ++ show x ++ ", " ++ show y ++ ", (/=))"
        ++ if not $ null cs then ", " ++ makeShow cs else ""
    strD = show d
    in "GC (AC3 " ++ strCon ++ " " ++ strD ++ " )"
```

We define a method to convert a graph into an instance of `GraphCol`, and vice versa. Note that graph colouring concerns *un*directed graphs while the Graph library concerns *directed* graphs. As a result, `g == ac3ToGraph $ convertGraphToAC3 g n` (for any $n > 0$) is NOT guaranteed to hold.

```
-- NOTE: The Graph library uses *directed* graphs.
--       We add both (x,y,/=) and (y,x,/=), as graph colouring concerns Undirected graphs.
-- Create an instance with colours [0..(n-1)]
convertGraphToAC3 :: Graph -> Int -> GraphCol
convertGraphToAC3 g n = let
    agents = vertices g
    constr = [(x,y, (/=)) | (x,y)<-edges g]
    in GC $ AC3
        (constr ++ reverseCons constr)
        -- In graph colouring, we want to check both X's domain to Y, and Y's to X.
        ((head agents, [0]) : [(a, [0..(n-1)]) | a<-tail agents])

-- Help function: If we have an edge (x,y), we need both (x,y, /=) and (y,x,/=) as
    constraints.
reverseCons :: [(a,b,c)] -> [(b,a,c)]
reverseCons = map (\ (a,b,c) -> (b,a,c))

ac3ToGraph :: GraphCol -> Graph
ac3ToGraph (GC (AC3 c d)) = let
    v = [a | (a,_)<-d]
    e = nub [ (a,b) | (a,b,_)<-c] -- If we originally had (x,y) AND (y,x) in our graph,
        then c contains each twice.
    in buildG (foldr min 0 v, foldr max minBound v) e




{-
-- TODO Remove?
ac3GetNcolours :: GraphCol -> Int
ac3GetNcolours (GC (AC3 _ (d':ds))) = (\(_,xs) -> foldr max 0 xs) d'
ac3GetNcolours (GC (AC3 _ d)) = foldr (\(_,xs) -> foldr max 0 xs) 0 d
-}
```

The actual main part of the programme, for Graph Colouring:

```
getGraphChoice :: IO Int
getGraphChoice = do
```

```haskell
    putStr "Choose one of the following options: \n\
           \1: Read in a graph from the terminal \n\
           \2: TODO \n"
    choice <- getLine
    case readMaybe choice of
      Nothing -> do
        putStrLn "Invalid choice, please try again."
        getGraphChoice
      Just n ->
        if n > 0 && n < 2 then return n else do
          putStrLn "Invalid choice, please try again."
          getGraphChoice

graphColMain :: IO ()
graphColMain = do
    choice <- getGraphChoice
    case choice of
        1 -> terminalGraph
        _ -> undefined

-- PRE: m <= n.
getEdges :: Int -> Int -> IO [Edge]
getEdges m n
  | m == n = return []
  | otherwise = do
      putStrLn $ "Edge " ++ show m
      x <- parseInput "Enter the first vertex: "
      y <- parseInput "Enter the second vertex: "

      rest <- getEdges (m+1) n
      return $ (x,y) : rest

terminalGraph :: IO ()
terminalGraph = do
  nVertices <- parseInput "Enter the number of vertices: "
  putStrLn $ "Okay, we number the vertices from 0 to " ++ show (nVertices-1)
  -- TODO: Error handling, eg. if nVertices <= 0 ?
  nEdges <- parseInput "Enter the number of edges: "
  eList <- getEdges 0 nEdges
  let graph = buildG (0, nVertices-1) eList
  nColours <- parseInput "Enter the number of colours: "
  let g = convertGraphToAC3 graph nColours
  putStrLn "We run AC3 on this instance."
  runGraph g


-- Given a graphcol instance, we run AC3 on it. If we have at least 1 solution (after back
    prop.),
--  show it to the user, and ask if they want to see all solutions.
runGraph :: GraphCol -> IO ()
runGraph (GC ac3Inst) = do
  let ac3Domain = ac3 ac3Inst
  if determineNoSol ac3Domain
    then putStrLn "AC3 has found an empty domain for at least 1 agent -> No solution"
    else do
      putStrLn "AC3 has at least 1 option for each agent."
      case findSolution ac3Inst of
        Nothing -> putStrLn "There is no solution based on the reduced AC3 input."
        Just sol -> do
          putStrLn $ "We have found a solution: " ++ show sol
          putStrLn "Do you want to find out how many different solutions we have? (Y/N) "
          choice <- getLine
          when (toUpper (head choice) == 'Y') $ do
            let allSols = findAllSolutions ac3Inst
            if length allSols == 1 then putStrLn "There is only 1 solution."
            else do -- it should not be possible to reach here if allSols = 0.
              putStrLn $ "There are " ++ show (length allSols) ++ " different solutions. \
                  nDo you want to see them? (Y/N) "
              choice2 <- getLine
              when (toUpper (head choice2) == 'Y') $ mapM_ print allSols
```

```haskell
-- Note: The graph we read in must be a CSV of an adjecency list;
-- https://hackage.haskell.org/package/graphite-0.10.0.1/docs/Data-Graph-Read.html
csvGraph :: IO ()
csvGraph = undefined
{-csvGraph = do
    putStrLn "Give a filepath to the graph you want to read in."
    filePath <- getLine
    x <- fromCsv filePath

    undefined
    --case fromCsv filePath of
    --    Left _ -> undefined
    --    Right _ -> undefined

-}
```

```haskell
module Scheduling where

import Text.Parsec
import Text.Parsec.String
import Control.Monad (replicateM)
import Data.List (elemIndex)
import AC3Solver
import Backtracking (findSolution)

type ClassAssignment = (Int, Int, Int)
dayNames :: [String]
dayNames = ["monday", "tuesday", "wednesday", "thursday", "friday"]

parseInt :: Parser Int
parseInt = do
  spaces
  n <- many1 digit
  return (read n)

parseInput :: String -> IO Int
parseInput prompt = do
  putStrLn prompt
  read <$> getLine

getNames :: String -> Int -> IO [String]
getNames prompt n = do
  putStrLn prompt
  replicateM n getLine

getUserInputs :: IO (Int, Int, Int, [String], [String], [String])
getUserInputs = do
  numClasses <- parseInput "Enter the number of classes:"
  classNames <- getNames "Enter class names:" numClasses
  numRooms <- parseInput "Enter the number of rooms:"
  roomNames <- getNames "Enter room names:" numRooms
  numTimeSlots <- parseInput "Enter the number of time slots per day:"
  timeSlotNames <- getNames "Enter time slot names:" numTimeSlots
  return (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames)

testUserInputs :: IO ()
testUserInputs = do
  (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames) <-
      getUserInputs

  putStrLn "\nCollected Inputs:"
  putStrLn $ "Number of Classes: " ++ show numClasses
  putStrLn $ "Class Names: " ++ show classNames
  putStrLn $ "Number of Rooms: " ++ show numRooms
  putStrLn $ "Room Names: " ++ show roomNames
  putStrLn $ "Number of Time Slots per Day: " ++ show numTimeSlots
  putStrLn $ "Time Slot Names: " ++ show timeSlotNames

checkSameDay :: ClassAssignment -> ClassAssignment -> Bool
checkSameDay (x,_,_) (y,_,_) = x == y
```

```haskell
checkBefore :: ClassAssignment -> ClassAssignment -> Bool
checkBefore (x,x2,_) (y,y2,_) = x == y && x2 + 1 == y2

checkAfter :: ClassAssignment -> ClassAssignment -> Bool
checkAfter (x,x2,_) (y,y2,_) = x == y && x2 == y2 + 1

checkDay :: Int -> ClassAssignment -> Bool
checkDay a (x,_,_) = x == a

checkTime :: Int -> ClassAssignment -> Bool
checkTime a (_,x,_) = x == a

checkRoom :: Int -> ClassAssignment -> Bool
checkRoom a (_,_,x) = x == a

filterDomains :: [Domain Int ClassAssignment] -> [(Agent Int, ClassAssignment -> Bool)] ->
    [Domain Int ClassAssignment]
filterDomains domainList conditions =
    [(agent, [v | v <- values, all (\(a, f) -> (a /= agent) || f v) conditions]) | (agent,
        values) <- domainList]

getConstraint :: [String] -> IO (Maybe [ConstraintAA Int ClassAssignment])
getConstraint classNames = do
  putStrLn "Enter a constraint (e.g., 'class1 is before class2' or 'class1 is the same day
      as class2'). Type 'Done' to finish:"
  input <- getLine
  if input == "Done" then return Nothing else do
    let parts = words input
    case parts of
      [class1, "is", "before", class2] -> Just <$> processConstraints class1 class2 "is
          before" classNames
      [class1, "is", "the", "same", "day", "as", class2] -> Just <$> processConstraints
          class1 class2 "is the same day as" classNames
      _ -> do
        putStrLn "Invalid input format"
        getConstraint classNames

processConstraints :: String -> String -> String -> [String] -> IO [ConstraintAA Int
    ClassAssignment]
processConstraints class1 class2 keyword classNames = do
  case (elemIndex class1 classNames, elemIndex class2 classNames) of
    (Just i, Just j) -> case keyword of
      "is before" -> return [(i, j, checkBefore), (j, i, checkAfter)]
      "is the same day as" -> return [(i, j, checkSameDay), (j, i, checkSameDay)]
      _           -> error "Invalid keyword"
    _ -> error "Invalid class names"

collectConstraints :: [String] -> IO [ConstraintAA Int ClassAssignment]
collectConstraints classNames = do
  let loop acc = do
        constraint <- getConstraint classNames
        case constraint of
          Nothing -> return acc
          Just cs  -> loop (cs ++ acc)
  loop []

getStartingValues :: [String] -> [String] -> [String] -> IO (Maybe (Agent Int,
    ClassAssignment -> Bool))
getStartingValues classNames roomNames timeslotNames = do
  putStrLn "Enter known values (e.g., 'class1 is in room3', 'class1 is at 11am' or 'class1
      is on monday'). Type 'Done' to finish:"
  input <- getLine
  if input == "Done" then return Nothing else do
    let parts = words input
    case parts of
      [class1, "is", "in", room] -> Just <$> processStartingValues class1 room "is in"
          classNames roomNames
      [class1, "is", "at", time] -> Just <$> processStartingValues class1 time "is at"
          classNames timeslotNames
      [class1, "is", "on", day] -> Just <$> processStartingValues class1 day "is on"
          classNames dayNames
      _ -> do
```

```
            putStrLn "Invalid input format"
            getStartingValues classNames roomNames timeslotNames

processStartingValues :: String -> String -> String -> [String] -> [String] -> IO (Agent
    Int, ClassAssignment -> Bool)
processStartingValues class1 value keyword classNames valueNames = do
  case (elemIndex class1 classNames, elemIndex value valueNames) of
    (Just i, Just j) -> case keyword of
      "is in" -> return (i, checkRoom j)
      "is at" -> return (i, checkTime j)
      "is on" -> return (i, checkDay j)
      _           -> error "Invalid keyword"
    _ -> error "Invalid name"

collectStartingValues :: [String] -> [String] -> [String] -> IO [(Agent Int,
    ClassAssignment -> Bool)]
collectStartingValues classNames roomNames timeslotNames = do
  let loop acc = do
        value <- getStartingValues classNames roomNames timeslotNames
        case value of
          Nothing -> return acc
          Just svs  -> loop (svs : acc)
  loop []

printSolution :: [String] -> [String] -> [String] -> [String] -> [(Agent Int,
    ClassAssignment)] -> IO ()
printSolution classNames days roomNames timeSlotNames list = putStrLn $ concat
  [classNames !! agent ++ " is scheduled on " ++ days !! dayId ++
    " in " ++ roomNames !! roomId ++ " at " ++ timeSlotNames !! timeId ++ ".\n"
  | (agent, (dayId, roomId, timeId)) <- list]

schedulingMain :: IO ()
schedulingMain = do
  (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames) <-
      getUserInputs
  constraints <- collectConstraints classNames

  let uniquenessConstraints = [(i, j, (/=)) | i <- [0..numClasses-1], j <- [0..numClasses
      -1], i /= j]
  let allConstraints = constraints ++ uniquenessConstraints
  let classDomains = [(i, [(d, t, r) | d <- [0..5], t <- [0..numTimeSlots-1], r <- [0..
      numRooms-1]]) | i <- [0..numClasses-1]]
  domainConditions <- collectStartingValues classNames roomNames timeSlotNames
  let filteredDomains = filterDomains classDomains domainConditions

  let possibleSolutions = ac3 AC3 { cons = allConstraints, domains = filteredDomains }

  let solution = findSolution AC3 { cons = allConstraints, domains = possibleSolutions }

  case solution of
    Nothing -> putStrLn "No solution found."
    Just sol -> printSolution classNames dayNames roomNames timeSlotNames sol
```

```
module Sudoku where

sudokuMain :: IO ()
sudokuMain = undefined
```

The constraints on any given cell is: 1. it must contain a distinct number than other cells in the same row, 2. it must contain a distinct number than other cells in the same column, 3. it must contain a distinct number than other cells in the same 3x3 box.

An 'Agent' can be represented by a cell, which in-turn can be represented as a tuple (i,j) where i is the row number and j is the column number.

```
-- Encode all possible cells in a 9x9 grid
allCells :: [(Int, Int)]
allCells = [(i,j) | i <- [1..9], j <- [1..9]]

sameRow :: (Int, Int) -> (Int, Int) -> Bool
```

```haskell
sameRow (x1,_) (y1,_) = x1 == y1

sameCol :: (Int, Int) -> (Int, Int) -> Bool
sameCol (_,x2) (_,y2) = x2 == y2

sameBox :: (Int, Int) -> (Int, Int) -> Bool
sameBox (x1,y1) (x2,y2) = (x1 - 1) `div` 3 == (x2 - 1) `div` 3 && (y1 - 1) `div` 3 == (y2 -
    1) `div` 3
-- for all x: (fst x - 1) in {0,1,2} (box row)
-- for all y: (snd y - 1) in {0,1,2} (box column)
-- when both the box row and box column are the same for x and y, they are in the same box.

-- All cells must obey the three constraints.
sudokuConstraints :: [ConstraintAA (Int, Int) Int]
sudokuConstraints =    [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameRow` j]
                    ++ [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameCol` j]
                    ++ [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameBox` j]

-- The domain will change depending on the puzzle being solved. Some cells should have
    fixed values.
-- Each cell can take on a value from 1 to 9.
sudokuDomains :: [Domain (Int, Int) Int]
sudokuDomains = [(i, [1..9]) | i <- allCells]

-- An empty Sudoku puzzle (no cells filled in)
sudokuMain :: AC3 (Int, Int) Int
sudokuMain = AC3 sudokuConstraints sudokuDomains
```

Creating an example puzzle that has a unique solution.

```haskell
-- Define the starting values
startingCellsUnique :: [Domain (Int, Int) Int]
startingCellsUnique = [
    ((1,3), [1]),
    ((1,5), [6]),
    ((1,9), [4]),
    ((2,1), [8]),
    ((2,4), [1]),
    ((2,6), [4]),
    ((2,8), [6]),
    ((3,2), [3]),
    ((3,7), [8]),
    ((3,9), [5]),
    ((4,1), [7]),
    ((4,3), [8]),
    ((4,5), [2]),
    ((4,9), [3]),
    ((5,2), [6]),
    ((5,3), [3]),
    ((5,7), [1]),
    ((5,8), [2]),
    ((5,9), [9]),
    ((6,5), [1]),
    ((7,1), [3]),
    ((7,7), [2]),
    ((7,9), [8]),
    ((8,1), [1]),
    ((8,3), [4]),
    ((8,5), [5]),
    ((8,6), [9]),
    ((8,7), [3]),
    ((9,2), [7]),
    ((9,4), [8]),
    ((9,6), [3]),
    ((9,7), [5]),
    ((9,8), [9]),
    ((9,9), [1])
    ]

-- Combine the starting values with the rest of the (empty) cells
sudokuExampleDomainUnique :: [Domain (Int, Int) Int]
sudokuExampleDomainUnique = startingCellsUnique ++ [(i, [1..9]) | i <- allCells, i `notElem
```

```
        ' map fst startingCellsUnique]

sudokuExampleUnique :: AC3 (Int, Int) Int
sudokuExampleUnique = AC3 sudokuConstraints sudokuExampleDomainUnique
```

Function to visualize the Sudoku board.

```
-- Visualize a Sudoku board based on current domains
visualizeSudoku :: [Domain (Int, Int) Int] -> String
visualizeSudoku domains' = unlines (
    horizontalLine : concatMap formatRow [1 .. 9]
  )
  where
    -- Get the domain for a specific cell
    getDomain i j =
        case [d | ((i', j'), d) <- domains', i' == i, j' == j] of
            (d:_) -> d
            [] -> [1..9]   -- Default domain if not specified

    -- Display a cell: single digit if domain has 1 element, empty otherwise
    cellValue i j =
        let domain' = getDomain i j
        in if length domain' == 1
            then show (head domain')
            else " "

    -- Horizontal line patterns
    horizontalLine = "+---+---+---+---+---+---+---+---+---+"
    thickLine = "+===+===+===+===+===+===+===+===+===+"

    -- Format a row with appropriate separators
    formatRow i =
        let rowStr = "| " ++ intercalate " | " [cellValue i j | j <- [1..9]] ++ " |"
            separator = if i 'mod' 3 == 0 && i /= 9 then thickLine else horizontalLine
        in [rowStr, separator]

-- Function to print the initial state of a Sudoku puzzle
printSudokuPuzzle :: AC3 (Int, Int) Int -> IO ()
printSudokuPuzzle (AC3 _ domains') = do
    putStrLn "Initial Sudoku puzzle:"
    putStrLn (visualizeSudoku domains')

-- Function to run and print a Sudoku solution
printSudokuSolution :: AC3 (Int, Int) Int -> IO ()
printSudokuSolution puzzle = do
    let solution = ac3 puzzle
    putStrLn (visualizeSudoku solution)
```

```
-- Compute the amount of reduction in domain size before and after applying AC-3

computeReduction :: AC3 (Int, Int) Int -> (Float, Float)
computeReduction puzzle = (fromIntegral originalSize/81, fromIntegral reducedSize/81)
  where
    originalSize = sum (map length (getDomains puzzle))
    reducedSize = sum (map (length . snd) (ac3 puzzle)) -- Using AC-3 algorithm to reduce
        the domains

-- Extract the domains of each cell from a sudoku puzzle
getDomains :: AC3 (Int, Int) Int -> [[Int]]
getDomains (AC3 _ domains') = map snd domains'
```

4 The Main file

5 Wrapping it up in an exectuable

We will now use the library form Section **??** in a program.

```haskell
module Main where

import Text.Read (readMaybe)

import Basics
import GraphCol
import Knapsack
import Scheduling
import Sudoku
import NQueens
import ZebraPuzzle

getChoice :: IO Int
getChoice = do
  putStr "Choose one of the following options: \n\
         \1: Graph Colouring \n\
         \2: N-Queens \n\
         \3: Scheduling \n\
         \4: Sudoku \n\
         \5: Zebra Puzzle \n"
  choice <- getLine
  case readMaybe choice of
    Nothing -> do
      putStrLn "Invalid choice, please try again."
      getChoice
    Just n ->
      if n > 0 && n < 6 then return n else do
        putStrLn "Invalid choice, please try again."
        getChoice

main :: IO ()
main = do
  putStrLn "Hello!"
  --print somenumbers
  --print (map funnyfunction somenumbers)
  --myrandomnumbers <- randomnumbers
  --print myrandomnumbers
  --print (map funnyfunction myrandomnumbers)
  --putStrLn "GoodBye"

  -- Get choice
  choice <- getChoice
  case choice of
    1 -> graphColMain
    2 -> nQueensMain
    3 -> schedulingMain
    4 -> sudokuMain
    5 -> zebraPuzzleMain
    _ -> undefined
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
```

```
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

6   The test file(s)

7   AC3 tests

```
module Main where

import AC3Solver
import Backtracking

import Data.Maybe
import Test.Hspec
import Test.QuickCheck

main :: IO ()
main = hspec $ do
  describe "AC3 Tests" $ do
    it "Example test" $
      ac3 exampleAC3 `shouldBe` [(4,[1,2]),(3,[0,1,2]),(2,[0,1,2]),(1,[0,1,2]),(0,[0])]
    it "Positive example (each agent has non-empty domain) - 1" $
      ac3 exampleAC3 `shouldNotSatisfy` determineNoSol
    it "Positive example (each agent has non-empty domain) - 2" $
      ac3 exampleAC3_2 `shouldNotSatisfy` determineNoSol
    it "Positive example (each agent has non-empty domain) - 3" $
      ac3 exampleAC3_GFG `shouldNotSatisfy` determineNoSol

    it "Positive example (at least 1 actual solution) - 1" $ do
      let newD = ac3 exampleAC3
      findSolution (AC3 (cons exampleAC3) newD) `shouldSatisfy` isJust
    it "Positive example (at least 1 actual solution) - 2" $ do
      let newD = ac3 exampleAC3_2
      findSolution (AC3 (cons exampleAC3_2) newD) `shouldSatisfy` isJust
    it "Positive example (at least 1 actual solution) - 3" $ do
      let newD = ac3 exampleAC3_GFG
      findSolution (AC3 (cons exampleAC3_GFG) newD) `shouldSatisfy` isJust

    it "Negative example (has no solution) - 1" $ do
      let newD = ac3 exampleAC3_bad
      findSolution (AC3 (cons exampleAC3_bad) newD) `shouldBe` Nothing
    it "Negative example (has no solution) - 2" $ do
      let newD = ac3 exampleAC3_triv
      findSolution (AC3 (cons exampleAC3_triv) newD) `shouldBe` Nothing
    it "Negative example (has no solution) - 3" $ do
      let newD = ac3 exampleAC3_no_solution
      findSolution (AC3 (cons exampleAC3_no_solution) newD) `shouldBe` Nothing

    -- The --coverage says these cases are never reached, but that is simply not true lol.
    --  It says this even with these cases, but that notwithstanding: according to the test
        report,
    --  we always get the otherwise case, which would seemingly point to us eventually
        reaching the
    --      [] = undefined case       \_(   )_/
    let xAgent = ("x", [1 :: Int])
    let yAgent = ("y", [2])
    let d = [xAgent, yAgent]
    it "Test popXy x==a" $ do
      popXy "x" "y" d `shouldBe` ([1], [2], [yAgent])
    it "Test popXy y==a" $ do
      popXy "y" "x" d `shouldBe` ([2], [1], [xAgent])
```

13

```
-- TEST CASES

exampleAC3 :: AC3 Int Int
exampleAC3 = let
    nColours = 3
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node. (TODO: for general
        encoding, if a vertex has no edges, assign an arbit colour.)
    in AC3 [ (a, (a+1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]] ((0, [0]) : [ (a, [0..
        nColours-1]) | a<-[1..nAgents-1]])


-- A graph is 2-colourable iff it is bipartite iff it has no cycles of odd length.
-- (Such as, this example which is a circle of even length.)
exampleAC3_2 :: AC3 Int Int
exampleAC3_2 = let
    nColours = 2
    nAgents = 6
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a-1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]]++[ (a, (a+1) `mod`
        nAgents, (/=)) | a<-[0..nAgents-1]])
            ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- NOT 2-colourable, as it has an odd cycle (circle of len 5).
exampleAC3_bad :: AC3 Int Int
exampleAC3_bad = let
    nColours = 2
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a-1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]]++[ (a, (a+1) `mod`
        nAgents, (/=)) | a<-[0..nAgents-1]])
            ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- NOT 1-colourable, as it has an edge.
exampleAC3_triv :: AC3 Int Int
exampleAC3_triv = let
    nColours = 1 -- can only be 1-colourable iff cons = [].
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a+1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]])
            ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- Example based on https://www.geeksforgeeks.org/3-coloring-is-np-complete/
-- IS 3-colourable.
exampleAC3_GFG :: AC3 String Int
exampleAC3_GFG = let
    nColours = 3 -- can only be 1-colourable iff cons = [].
    agentsA = ["v", "w", "u", "x"]
    agentsB = [s++"'" | s<-agentsA]
    agents = agentsA ++ agentsB -- does NOT include "B"

    bCons = [("B", a, (/=)) | a<-agents]
    outsideCons = [ (a, a++"'", (/=)) | a<-agentsA ]
    reverseCons = map (\ (a,b,c) -> (b,a,c))
    in AC3 (bCons ++ reverseCons bCons ++
                outsideCons ++ reverseCons outsideCons)
            ( ("B", [0]) : [ (a, [0..nColours-1]) | a <- agents])

-- A problem that should have no solutions
exampleAC3_no_solution :: AC3 Int Int
exampleAC3_no_solution = let
    domains_no_sol = [(0, [1,2]), (1, [1,2]), (2, [1,2])]
    constraints_no_sol = [(0, 1, (/=)), (1, 2, (/=)), (0,2, (/=))]
    in AC3 constraints_no_sol domains_no_sol
```

8   Conclusion

Finally, we can see that [**?**] is a nice paper.

References

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.