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`

```
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
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.

```haskell
-- 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'
```

Giga mess, lots of TODOs and optimisations etc., but this seems to work for the simple AC3 test cases I have below:

```
module AC3_Mess where

-- TODO: Use sequence to implement a priority queue-type thing for the constraints?
--import Data.Sequence (Seq, ViewR(..), ViewL(..), (<|), (|>), (><))
--import qualified Data.Sequence as Seq -- https://hackage.haskell.org/package/containers
    -0.8/docs/Data-Sequence.html

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

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] }


    --deriving (Eq,Ord,Show) -- requires this from a & b...
    -- Plus, I doubt we can use this given constraint is a function...

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

-- Return the elements of xs for which there exist a y \in ys, such that c x y holds.
checkDomain :: [a] -> [a] -> Constraint a -> [a]
checkDomain xs ys c = filter (`checkX` ys) xs where
    checkX _ [] = False
    checkX x' (y:ys') = c x' y || checkX x' ys'
```

```
-- 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.
checkDomain2 :: [a] -> [a] -> Constraint a -> Writer String [a]
checkDomain2 [] _ _ = return []
checkDomain2 (x:xs) ys c = do
    rest <- checkDomain2 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.

-- Then we can use with this with: runWriter $ checkDomain2 xs ys c
-- newX = fst result, output = snd result.
-- If output == "", then domain unchanged, and we can simply forget the constraint.
-- Else, look through original set of constraints, and re-add all constraints for ys.
```

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)
```

TODO

```
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
    --xDomain = head [b | (a,b)<-d, x==a ]
    --yDomain = head [b | (a,b)<-d, y==a ]
    (newX, str) = runWriter $ checkDomain2 xDomain yDomain c
    -- In a lens, we could do this with "modify (\ (a,_) -> (a, newX))"
    newDomains = (x, newX) : alteredD
    --newDomains = (x, newX) : ((x, xDomain) `delete` d)
    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
```

And some test cases, for the graph colouring problem. If I'm not mistaken, G is n-colourable

(for given n, assuming correct definition of instance) iff AC3 outputs at least 1 option in the domain of each Agent.

```
-- 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
```

2 The test file(s)

3 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])
```

```
-- 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
```

4 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.