Functional Programming & Constraint Programming

Flip Lijnzaad and Jasper Stammes

Tuesday 1st February, 2022

Abstract

This is where the abstract should go

Contents

1 Constraint satisfaction problems

```
module CSP where

-- type definitions based on formal definition of CSP in book
type Variable   = Int
type Value      = Int
type Domain     = (Variable, [Value])
type Arc        = (Variable, Variable)
type Constraint = ( Arc, [(Value, Value)] )
data Problem    = CSP { variables   :: [Variable]
                      , domains     :: [Domain]
                      , constraints :: [Constraint] } deriving Show
```

2 Arc consistency

3 The AC-3 algorithm

This functional implementation of the AC-3 algorithm is based on the imperative pseudocode in [RN09, p. 209].

```
module AC3 where

import CSP
import Data.List
-- implementation of the AC-3 function, recursive version of the pseudocode in
-- the book; calls ‘revise‘ helper function. the book version passes a queue
-- of arcs; we use a list of constraints, since those contain the arcs
ac3 :: (Problem, Bool, [Constraint]) -> (Problem, Bool, [Constraint])
-- if the Bool flag is False, the CSP has no solution, so stop the recursion
ac3 (p, False, _) = (p, False, [])
-- if the arc queue is empty, stop the recursion and return True
ac3 (p, True, []) = (p, True,  [])
-- else, perform body of the ‘while‘ loop
ac3 (p@(CSP vars doms cons), True, ((varX, varY), rel):xs) =
  if strongLookup varX doms == newXDomain
    -- if after revising, the domain of x stays the same,
    -- continue with the next arc in the queue and pass whether newXDomain is nonempty
    then ac3 (p, not $ null newXDomain, xs)
    -- if the domain of x has changed, need to add x’s neighbors to queue
    else ac3 (CSP vars newDoms cons, True, newQueue)
  where
    newXDomain = revise ((varX, varY), rel) (strongLookup varX doms) (strongLookup varY
        doms)
    -- delete x’s old domain and add x’s new domain to the list of domains
    newDoms    = newXDomain : delete (strongLookup varX doms) doms
    -- append to the arc queue xs the neighbors of x by filtering on (_, x)
    newQueue   = xs ++ filter (\(arc, _) -> snd arc == varX) cons

-- perform lookup and drop the Maybe
strongLookup :: Variable -> [Domain] -> Domain
strongLookup x v = let (Just y) = lookup x v in (x,y)

-- implementation of the revise function of the pseudocode in the book
revise :: Constraint -> Domain -> Domain -> Domain
-- trivial case: if there are no constraints, pass a domain with empty list of values
revise (_ , []) (varX,  _) _ = (varX, [])
-- if the domain for x is empty, pass domain with empty list of values for x
revise (_, _) (varX, []) _ = (varX, [])
-- else, perform body of the ‘for each‘ loop
revise (arc, rel) (varX, x:xs) (varY, ys) =
  if any (\y -> (x, y) ‘elem‘ rel) ys
    -- if there is a value y in ys that satisfies the contraint between x and y,
```

```
    -- add x to the domain and continue
    then prependToSnd x (revise (arc, rel) (varX, xs) (varY, ys))
    -- if there is none, continue without adding x
    else revise (arc, rel) (varX, xs) (varY, ys)
-- test case : revise ((100,101),[(x,y)| x<-[1..4], y<-[1..4], x==y]) (100,[1..3])
    (101,[2..4])

-- prepend a value to the value list of a domain (the second argument of the tuple)
prependToSnd :: Value -> Domain -> Domain
prependToSnd x (varX, xs) = (varX, x:xs)

-- since ac3 outputs a CSP including all of the constraints, we use this to return only the
     domain. Note that the problem has a unique solution if all problems have size 1

ac3domain :: [Variable] -> [Domain] -> [Constraint] -> [Domain]
ac3domain vars doms cons = let (CSP _ y _, _, _) = ac3 (CSP vars doms cons, True, cons) in
    sortBy (\(a,_) (b,_) -> compare a b) y
```

4 Sudokus

Sudokus are a well-known constraint satisfaction problem: each square of the 9×9 grid is constrained by the squares in the same row, the same column, and the same 3×3 block. In order to use the AC-3 algorithm on sudokus, the sudoku first needs to be represented as a constraint satisfaction problem (see Section 1). In order to do so, the variables, domains and constraints of the problem need to be specified.

```
module Sudoku where

import CSP
import AC3
import Data.Char         -- for using "digitToInt"
import Data.Maybe        -- for using "fromJust"
import Control.Monad     -- for using "when"
```

We have chosen to represent the 81 squares of the grid as numbers between 0 and 80.

say something about being in line with the CSP definition

The domain of each empty square of a sudoku is $\{1, \ldots, 9\}$; the domain of a square filled with some x is $\{x\}$. Since a `Domain` in our `CSP` definition also consists of the variable's `Int`, the following code also computes the 'index' of the square as a number between 0 and 80.

say something about the Python code and its formatting: we input the sudoku we want to solve as a string where empty cells are zeroes, a zero means the starting domain can be anything in $\{1, \ldots, 9\}$, if the cell is given its domain has just that element

```
generateSudokuDomains :: [Value] -> [Domain]
generateSudokuDomains [] = []
generateSudokuDomains (x:xs)
    | x == 0     = (80 - length xs, [1..9]):generateSudokuDomains xs
    | otherwise = (80 - length xs, [x]   ):generateSudokuDomains xs
```

Arguably the most interesting part now is how the constraints for each variable are generated. To be able to formulate the constraints in an intuitive way, the function `varToCoords` takes a variable and returns a tuple of its x- and y-coordinates within the 9×9 grid. `varToCoords` functions as a wrapper around the `varGrid` to eliminate some duplicate code.

```
varGrid :: [(Variable, (Value, Value))]
varGrid = zip [0..80] [ (i,j) | i <- [0..8], j <- [0..8] ]
varToCoords :: Variable -> (Value, Value)
varToCoords n = fromJust $ lookup n varGrid
```

Now, the function `generateSudokuConstraints` takes the list of all variables of the sudoku, and returns the list of constraints for the sudoku. It creates this list of constraints by working through the list of variables one by one and generating all constraints for each variable. As said before, each square on the grid is constrained by its row, column and 3×3 block. So a variable n is a member of all arcs $\langle n, x \rangle$ where x is a variable in the same row, column or block. The allowable values for the pair $\langle n, x \rangle$ are then all $y_1, y_2 \in \{1, \ldots, 9\}$ such that $y_1 \neq y_2$.

```
generateSudokuConstraints :: [Variable] -> [Constraint]
generateSudokuConstraints [] = []
generateSudokuConstraints (n:xs) =
  map (\x -> ( (n,x), [(y1,y2) | y1 <- [1..9], y2 <- [1..9], y1 /= y2] ) )
```

The row, column and block constraints are dependent on the position of the variable n within the grid. The following code fragment determines the variables x with which n is participating in a constraint. The variables in the same row as n have the same x-coordinate, and the variables in the same column as n have the same y-coordinate. To obtain the variables in the same 3×3 block as n, we check if the x-coordinates of n and m are the same when divided by 3; we do the same for the y-coordinates.

```
    (
      -- rows
        [m | m <- [0..80], m /= n,
              fst (varToCoords m) == fst (varToCoords n)]
      -- columns
      ++ [m | m <- [0..80], m /= n,
              snd (varToCoords m) == snd (varToCoords n)]
      -- blocks
      ++ [m | m <- [0..80], m /= n,
              fst (varToCoords m) /= fst (varToCoords n),
              snd (varToCoords m) /= fst (varToCoords n),
              fst (varToCoords m) `div` 3 == fst (varToCoords n) `div` 3,
              snd (varToCoords m) `div` 3 == snd (varToCoords n) `div` 3]
    )
      ++ generateSudokuConstraints xs
```

The list comprehension contains the Boolean condition $m \neq n$ to ensure that there will not be an arc $\langle n, n \rangle$ in the constraints, since there will be no assignment that satisfies the constraint $n \neq n$. Moreover, the list comprehension for the block constraints ensures that variables in the same row or column are ignored, since those have already been taken into account.

The `printSudoku` function takes the list of `Domain`s of a sudoku and prints the (partially) solved sudoku in a readable format using spaces and newlines. If the list of possible values for a variable only contains one element, this element may be printed; if it does not, then the value of that variable is as of yet undetermined and an underscore is printed to indicate this.

```
printSudoku :: [Domain] -> IO ()
printSudoku [] = putStr ""
printSudoku ((n, val@(value:_)):xs) =
  do
    -- put the number there if determined, else _
    putStr (if val == [value] then show value else "_")
    -- put spaces between different blocks
```

```
    when (n `mod` 3  == 2)  (putStr " ")
    -- put newlines at the end of rows
    when (n `mod` 9  == 8)  (putStr "\n")
    -- put extra newlines to vertically separate blocks
    when (n `mod` 27 == 26) (putStr "\n")
    do printSudoku xs
-- (to avoid warning about non-exhaustive cases)
printSudoku _ = putStr ""
```

```
-- solves the available sudoku in "sudoku.txt" in the "sudoku/" subdirectory
solveSudokuFromFile :: IO ()
solveSudokuFromFile = do
  sudokuString <- readFile "sudoku/sudoku.txt"
  -- make the string into a list of Ints
  let values = map digitToInt sudokuString
  -- solve the sudoku and print it
  do printSudoku $ ac3domain [0..80] (generateSudokuDomains values) (
      generateSudokuConstraints [0..80])
```

5 Conclusion

5.1 Further improvements and research

The double constraints could be eliminated from the sudoku CSP definition by using unordered pairs

References

[RN09] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.