Functional Programming & Constraint Programming

Flip Lijnzaad and Jasper Stammes

Thursday 3rd February, 2022

Abstract

This is where the abstract should go

Contents

1 Constraint satisfaction problems

A constraint satisfaction problem (CSP) is a triple $\langle X, D, C \rangle$ where

- X is a set of variables $\{x_1, \ldots, x_n\}$;
- D is a set of domains $\{D_1, \ldots, D_n\}$. Each domain is a set of values for a variable;
- C is a set of constraints on the domains of the variables.

A constraint is a pair $\langle scope,\ relation \rangle$, where the *scope* is a tuple of the variables that participate in the constraint, and the *relation* is a set of tuples determining the allowed combinations of values for the variables in the *scope*.

Any constraint with a finite scope can be reduced to a set of binary constraints [RN09, p. 206]; this greatly simplifies the representation of constraints. We call the scope of a binary constraint an *arc*.

In our Haskell implementation of CSPs, we define `type`s and `newtype`s for variables, values, domains, arcs, constraints and CSPs. In the formal definition of a CSP, which domains correspond to which variables is indicated by their subscripted indices matching. In our implementation, we chose to represent a domain as a tuple of the variable it pertains to, together with the list of possible values, e.g. $\langle x_1, D_1 \rangle$. Since then the set of domains already contains the variables as well, there is no need for including the set of variables in the definition of a CSP. Therefore, in our implementation a CSP is just the pair $\langle D, C \rangle$.

```
{-# LANGUAGE
  GeneralizedNewtypeDeriving
#-}

module CSP where

newtype Variable = Var { getVar :: Int } deriving (Eq, Show, Ord, Num)
newtype Value    = Val { getVal :: Int } deriving (Eq, Ord, Num)
instance Show Value where
  show x = show (getVal x)
type Domain      = (Variable, [Value])
type Arc         = (Variable, Variable)
type Constraint  = ( Arc, [(Value, Value)] )
data Problem     = CSP { domains     :: [Domain]
                       , constraints :: [Constraint] }
```

2 Arc consistency

A variable in a CSP is *arc-consistent* if every value in its domain satisfies the variable's binary constraints. A CSP is arc-consistent if every variable in it is arc-consistent with every other variable. Arc consistency is a desirable property of a CSP, since it restricts and thus minimizes the domains of the CSP's variables. An arc-consistent CSP is not necessarily a solved CSP; a CSP is solved if all constraints are satisfied by any combination of values the variables can take on. A CSP does not have a solution if one variable has an empty domain (i.e. no possible values it can take on).

The AC-3 algorithm reduces a CSP to its arc-consistent version. The algorithm return true if such an arc-consistent version exists, and it returns false if at any point a variable has an empty domain, i.e. the CSP has no solution.

3 The AC-3 algorithm

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

The implementation makes use of the auxiliary function `getVarDomain`. When given a variable and a list of domains, it returns the domain of that variable. It performs `lookup` on the list of domains and returns not only the second argument of the relevant tuple (which is a list of values), but the whole tuple (the whole domain). Moreover, it uses `fromJust` to strip the domain of its `Just`. Generally, this is not safe to do, but in this implementation we are certain that the `lookup` will never return `Nothing`: the use of `getVarDomain` is restricted to cases where we are certain that the variable to look up is actually present in the list.

```
module AC3 where

import CSP
import Data.List
import Data.Maybe -- for using "fromJust"

getVarDomain :: Variable -> [Domain] -> Domain
getVarDomain var doms = let dom = fromJust $ lookup var doms in (var, dom)
```

The `ac3` function takes as input a tuple containing the full CSP, a Boolean flag and a queue of constraints. The Boolean flag is the true or false that the algorithm returns as described in Section 2. The queue of constraints more or less functions as a to-do list, containing the arcs of which the consistency needs to be checked still. When first calling the function, this queue contains all constraints of the CSP; when the queue of constraints is empty, the CSP is arc-consistent. If at any point during the recursion the Boolean flag is set to false, this means that the domain of a variable is empty and the CSP has no solution. In this case, there is no point in continuing the recursion, so it is halted.

```
ac3 :: (Problem, Bool, [Constraint]) -> (Problem, Bool, [Constraint])
ac3 (p, False, _) = (p, False, [])
ac3 (p, True, []) = (p, True,  [])
```

In the recursive case, the first constraint $C = \langle\langle x, y\rangle, R\rangle$ in the constraint queue is considered. The arc $\langle x, y\rangle$ of C is then 'revised' based on this constraint: the domain of x is restricted to those values that satisfy the constraint C. More specifically, the new domain for x, `newXDomain`, is such that those $x' \in D_x$ are kept for which $\exists y' \in D_y$ such that $\langle x', y'\rangle \in R$.

If this revision did not change the domain of x (i.e., $\langle x, y\rangle$ was already arc-consistent), then the recursion continues with the CSP unchanged, with the flag still set to `True` (because no domains were changed) and with rest of the queue of constraints.

If the revision *did* change the domain of x, then this may cause changes in the domains of the 'neighbors' of x: the arcs of which x is the second argument. To propagate these changes, the neighbors of x are added to the queue (`newQueue`). The domain list is updated (`newDoms`) by deleting the old domain of x from it and adding the new domain of x to it. The new Boolean flag is whether or not x's new domain is empty.

```
ac3 (p@(CSP doms cons), True, ((x, y), rel):queue) =
  if getVarDomain x doms == newXDomain
    then ac3 (p, True, queue)
    else ac3 (CSP newDoms cons, not $ null $ snd newXDomain, newQueue)
  where
    newXDomain = ( x, [ x' | x' <- xvals, any (\y' -> (x', y') `elem` rel) yvals ] ) where
      xvals = snd $ getVarDomain x doms
      yvals = snd $ getVarDomain y doms
    newDoms  = newXDomain : delete (getVarDomain x doms) doms
    newQueue = queue ++ filter (\(arc, _) -> snd arc == x) cons
```

```
-- 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 :: [Domain] -> [Constraint] -> [Domain]
ac3domain doms cons =
  let (CSP y _, _, _) = ac3 (CSP doms cons, True, cons) in
  sortBy (\(a,_) (b,_) -> compare a b) y
-- do something about returning False!
```

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    = (Var (80 - length xs), map Val [1..9]) : generateSudokuDomains xs
    | otherwise = (Var (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, (Int, Int))]
varGrid = zip (map Var [0..80]) [ (i,j) | i <- [0..8], j <- [0..8] ]
varToCoords :: Variable -> (Int, Int)
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 <- map Val [1..9], y2 <- map Val [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 <- map Var [0..80], m /= n,
              fst (varToCoords m) == fst (varToCoords n)]
      -- columns
      ++ [m | m <- map Var [0..80], m /= n,
              snd (varToCoords m) == snd (varToCoords n)]
      -- blocks
      ++ [m | m <- map Var [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 (getVar n `mod` 3  == 2)  (putStr " ")
    -- put newlines at the end of rows
    when (getVar n `mod` 9  == 8)  (putStr "\n")
    -- put extra newlines to vertically separate blocks
    when (getVar n `mod` 27 == 26) (putStr "\n")
    do printSudoku xs
-- (to avoid warning about non-exhaustive cases)
printSudoku _ = putStr ""
```

add explanations here

```
-- 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 (Val . digitToInt) sudokuString
  -- solve the sudoku and print it
  do printSudoku $ ac3domain (generateSudokuDomains values) (generateSudokuConstraints (map
       Var [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.