My Report

Bo Flachs & Wessel Kroon

Wednesday 26th January, 2022

Abstract

Contents

1 Introduction

2 Inquisitive Semantics

3 InqB in Haskell

In this section we discuss the implementation of InqB in Haskell.

```
module InqB where

import Data.List

-- Type declarations of the models
type World        = Int
type Universe     = [World]
type Individual   = String

-- Type declarations for variables
type Var          = String
type Vars         = [Var]

data Varual       = Indv Individual | Var Var
        deriving (Eq, Ord, Show)

type Domain       = [Individual]
type UnRelation   = [(World, [Individual])]
type BiRelation   = [(World, [(Individual, Individual)])]
type TertRelation = [(World, [(Individual, Individual, Individual)])]

data Model = Mo { universe :: Universe
                , dom :: Domain
                , unRel :: [UnRelation]
                , biRel :: [BiRelation]
                , tertRel :: [TertRelation] }
        deriving (Eq, Ord, Show)

-- Type declarations for Propositions
type Prop     = [[World]]
type InfState = [World]

-- Type declarations for formulas
data Form = UnR UnRelation Varual
          | BinR BiRelation Varual Varual
          | TertR TertRelation Varual Varual Varual
          | Neg Form | Con Form Form | Dis Form Form
          | Impl Form Form
          | Forall Var Form | Exists Var Form
          deriving (Eq, Ord, Show)

-- Functions working on formulas
powerset :: [a] -> [[a]]
powerset []   = [[]]
powerset (x:xs) = powerset xs ++ map (x:) (powerset xs)

nonInq :: Form -> Form
nonInq = Neg . Neg

nonInf :: Form -> Form
nonInf f = Dis f $ Neg f
```

```
absPseudComp :: Model -> Prop -> Prop
absPseudComp m p = powerset $ universe m \\ (nub . concat) p

closeDownward :: [[World]] -> Prop
closeDownward = nub . concatMap powerset

myProp1 :: Prop
myProp1 = closeDownward [[1,2]]

myProp2 :: Prop
myProp2 = closeDownward [[1,3]]

relPseudComp :: Model -> Prop -> Prop -> Prop
relPseudComp m p q = filter (all (\t -> t `notElem` p || t `elem` q) .
    powerset )
                                  $ powerset $ universe m

substitute :: Individual -> Var -> Form -> Form
substitute d x (UnR r i)
                      | Var x == i  = UnR r (Indv d)
                      | otherwise   = UnR r i
substitute d x (BinR r i1 i2)       = BinR r (head varuals) (varuals !! 1)
                      where varuals = map (\i -> if Var x == i then Indv d
                          else i) [i1, i2]
substitute d x (TertR r i1 i2 i3)   = TertR r (head varuals) (varuals !! 1) (
    varuals !! 2)
                      where varuals = map (\i -> if Var x == i then Indv d
                          else i) [i1, i2, i3]
substitute d x (Neg f)              = Neg $ substitute d x f
substitute d x (Con f1 f2)          = Con (substitute d x f1) (substitute d x
    f2)
substitute d x (Dis f1 f2)          = Dis (substitute d x f1) (substitute d x
    f2)
substitute d x (Impl f1 f2)         = Impl (substitute d x f1) (substitute d x
     f2)
substitute d x (Forall y f)         = Forall y $ substitute d x f
substitute d x (Exists y f)         = Exists y $ substitute d x f

-- Helper function
getString :: Varual -> String
getString (Indv i) = i
getString (Var v)  = v

toProp :: Model -> Form -> Prop
toProp _ (UnR r i )         = closeDownward [[x |(x, y) <- r, getString i `
    elem` y]]
toProp _ (BinR r i1 i2)     = closeDownward [[x |(x, y) <- r, (getString i1,
    getString i2) `elem` y]]
toProp _ (TertR r i1 i2 i3) = closeDownward [[x |(x, y) <- r, (getString i1,
    getString i2, getString i3) `elem` y]]
toProp m (Neg f)            = absPseudComp m (toProp m f)
toProp m (Con f1 f2)        = toProp m f1 `intersect` toProp m f2
toProp m (Dis f1 f2)        = toProp m f1 `union` toProp m f2
toProp m (Impl f1 f2)       = relPseudComp m (toProp m f1) (toProp m f2)
-- Foldl1 has no base case so can only be applied to non-empty lists. We have
    a theoretical guarantee
-- that this is the case in the following.
toProp m (Forall x f)       = foldl1 intersect [ p | d <- dom m, let p =
    toProp m $ substitute d x f ]
toProp m (Exists x f)       = (nub . concat) [ p | d <- dom m, let p = toProp
    m $ substitute d x f ]

strictSubset :: InfState -> InfState -> Bool
strictSubset x y | null (x \\ y) && x /= y = True
                 | otherwise               = False

alt :: Model -> Form -> [InfState]
alt m f = sort [x | x <- p, not (any (strictSubset x) p)]
      where p = toProp m f
```

```
info :: Model -> Form -> InfState
info m f = sort . nub . concat $ toProp m f
```

4 Example models

In this section we create example models

```
module Examples where

import InqB

myR :: UnRelation
myR = [(1,["a","b"]), (2,["a"]), (3,["b"]), (4,[])]

myVars :: Vars
myVars = ["x", "y", "z"]

myModel :: Model
myModel = Mo
    -- Universe
    [1, 2,
     3, 4]
    -- Domain
    ["a", "b"]
    -- Unary relations
    [myR]
    -- BiRelation
    []
    -- TertRelation
    []
```

5 Model Checker

In this section we discuss the model checker

```
module Main where

import InqB
import Examples
import Data.List

main :: IO()
main = do putStrLn "Hello!"

-- Model checker
supportsProp :: InfState -> Prop -> Bool
supportsProp s p = s `elem` p

supportsForm :: Model -> InfState -> Form -> Bool
supportsForm m s f = supportsProp s $ toProp m f

testExample :: Bool
testExample = supportsForm myModel [1,2] (UnR myR (Indv "a"))

isInquisitive :: Model -> Form -> Bool
```

```
isInquisitive m f = sort (toProp m f) /= (sort . powerset) (info m f)

isInformative :: Model -> Form -> Bool
isInformative m f = (sort . universe) m /= sort (info m f)

isTautology :: Model -> Form -> Bool
isTautology m f = (sort. powerset . universe) m == sort (toProp m f)

entails :: Model -> Form -> Form -> Bool
entails m f1 f2 = all (`elem` p2) p1 where
             p1 = toProp m f1
             p2 = toProp m f2

isEquivalent :: Model -> Form -> Form -> Bool
isEquivalent m f g = toProp m f == toProp m g

makesTrue :: Model -> World -> Form -> Bool
makesTrue m w f = [w] `elem` toProp m f
```

6 Simple Tests

In this section we use QuickCheck to test some theorems from los bookos.

```
module Main where

import InqB
-- import InqB()
import Test.QuickCheck

main :: IO()
main = do putStrLn "Hello"

myIndividuals :: Domain
myIndividuals = ["a","b","c"]

myRelation :: UnRelation
myRelation = [(1, ["a"])]

-- NOg niet correct
instance Arbitrary Form where
  arbitrary = sized randomForm where
    randomForm :: Int -> Gen Form
    randomForm 0 = pure $ UnR myRelation "a" --UnR <$> elements myIndividuals
    randomForm n = oneof
      [ pure $ UnR myRelation "a" --Prp  <$> elements myAtoms
      , Neg  <$> randomForm (n `div` 2)
      , Con  <$> randomForm (n `div` 2)
             <*> randomForm (n `div` 2)
      , Dis  <$> randomForm (n `div` 2)
             <*> randomForm (n `div` 2)
      , Impl <$> randomForm (n `div` 2)
             <*> randomForm (n `div` 2)
      ]

trivialTest :: Form -> Bool
trivialTest _ = True
```

7 Conclusion

[Knu11]

References

[Knu11] Donald E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley Professional, 2011.