My Report

Bo Flachs & Wessel Kroon

Friday 28$^{\text{th}}$ January, 2022

Abstract

Contents

1 Introduction

2 Inquisitive Semantics

3 InqB in Haskell

3.1 Models

In this subsection we discuss the implementation of models.

```haskell
module InqBModels where

import HelperFunctions
import Test.QuickCheck

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

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]


myWorlds :: [World]
myWorlds = [1..4]

myIndiviuals :: [Individual]
myIndiviuals = ["a","b","c","d"]

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

instance Arbitrary Model where
  arbitrary = randomModel where
    randomModel :: Gen Model
    randomModel = Mo <$> u <*> d <*> ur <*> br <*> tr
      where u = elements $ powerset myWorlds
            d = sublistOf myIndividuals -- elements $ (filter (not . null) .
                powerset) myIndiviuals
            ur = zip <$> d <*> d
            br = pure []
            tr = pure []
```

3.2 Syntax

In this subsection we discuss the implementation of the syntax of InqB in Haskell.

```
module InqBSyntax where

import InqBModels
-- import Test.QuickCheck

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

-- Call this terms
data Term       = Indv Individual | Var Var
        deriving (Eq, Ord, Show)

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


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

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

{--- Define data ModelWithForm =  MWF (Model, Form)
-- Create arbitrary instance for that

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

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

instance Arbitrary Form where
  arbitrary = sized randomForm where
    randomForm :: Int -> Gen Form
    randomForm 0 = UnR <$> elements (unRel myModel')
                       <*> elements (map Var (dom myModel'))
    randomForm n = oneof
      [ UnR   <$> elements (unRel myModel')
              <*> elements (map Var (dom myModel'))
      , BinR  <$> elements (biRel myModel')
              <*> elements (map Var (dom myModel'))
              <*> elements (map Var (dom myModel'))
      , TertR <$> elements (tertRel myModel')
              <*> elements (map Var (dom myModel'))
              <*> elements (map Var (dom myModel'))
```

```
            <*> elements (map Var (dom myModel'))
    , 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)
    ]
    -}
```

3.3 Semantics

In this subsection we discuss the implementation of the semantics in Haskell.

```
module InqBSemantics where

import Data.List
import InqBModels
import InqBSyntax
import HelperFunctions


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


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 terms) (terms !! 1)
                      where terms = map (\i -> if Var x == i then Indv d else
                          i) [i1, i2]
substitute d x (TertR r i1 i2 i3)   = TertR r (head terms) (terms !! 1) (terms
     !! 2)
                      where terms = 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)
                    | x == y        = Forall y f
                    | otherwise     = Forall y $ substitute d x f
substitute d x (Exists y f)
                    | x == y        = Exists y f
                    | otherwise     = Exists y $ substitute d x f

getString :: Term -> String
getString (Indv i) = i
getString (Var v)  = v

toProp :: Model -> Form -> Prop
```

```haskell
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)
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 ]


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
```

3.4 Helper functions

In this subsection we discuss some helper functions that we implemented

```haskell
module HelperFunctions where

import Data.List

powerset :: [a] -> [[a]]
powerset []   = [[]]
powerset (x:xs) = powerset xs ++ map (x:) (powerset xs)


strictSubset :: Eq a => [a] -> [a] -> Bool
strictSubset x y | null (x \\ y) && x /= y = True
                 | otherwise               = False

closeDownward :: Eq a => [[a]] -> [[a]]
closeDownward = nub . concatMap powerset
```

3.5 Model Checker

In this subsection we discuss the implementation of the syntax of model checker in Haskell.

```haskell
module ModelChecker where

import InqBModels
import InqBSyntax
import InqBSemantics
import Examples

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

```
-- 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

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

4 Example models

In this section we create example models

```
module Examples where

import InqBModels
import InqBSyntax

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
    []
myR2 :: UnRelation
myR2 = [(1,["a","b"]), (2,["a,b"]), (3,[]), (4,[])]

myBiR :: BiRelation
myBiR = [(1,[("a","a"),("b","b")]),
         (2,[("a","a"),("b","b")]),
         (3,[("c","c"),("b","b")]),
         (4,[("a","a"),("d","d")])]

myTertR :: TertRelation
myTertR = [(1,[("a","a","b"),("b","b","c")]),
         (2,[("a","a","d"),("b","b","c")]),
         (3,[("c","a","c"),("d","b","b")]),
         (4,[("b","a","a"),("a","d","d")])]

-- myVars2 :: Vars
-- myVars2 = ["x", "y", "z"]

myModel2 :: Model
myModel2 = Mo
    -- Universe
    [1, 2,
     3, 4]
```

```haskell
    -- Domain
    ["a", "b"]
    -- Unary relations
    [myR, myR2]
    -- BiRelation
    [myBiR]
    -- TertRelation
    [myTertR]


form1 :: Form
form1 = UnR myR (Indv "a")

form2 :: Form
form2 = UnR myR (Indv "b")

form3 :: Form
form3 = Dis (UnR myR (Indv "a")) (UnR myR (Indv "b"))

form4 :: Form
form4 = Neg (UnR myR (Indv "a"))

form5 :: Form
form5 = Neg (Dis (UnR myR (Indv "a")) (UnR myR (Indv "b")))

form6 :: Form
form6 = nonInq (Dis (UnR myR (Indv "a")) (UnR myR (Indv "b")))

form7 :: Form
form7 = nonInf (UnR myR (Indv "a"))

form8 :: Form
form8 = nonInf (UnR myR (Indv "b"))

form9 :: Form
form9 = nonInf (Dis (UnR myR (Indv "a")) (UnR myR (Indv "b")))

form10 :: Form
form10 = nonInf (nonInq (Dis (UnR myR (Indv "a")) (UnR myR (Indv "b"))))

form11 :: Form
form11 = Con (UnR myR (Indv "a")) (UnR myR (Indv "b"))

form12 :: Form
form12 = Con (nonInf (UnR myR (Indv "a"))) (nonInf (UnR myR (Indv "b")))

form13 :: Form
form13 = Impl (UnR myR (Indv "a")) (UnR myR (Indv "b"))

form14 :: Form
form14 = Impl (UnR myR (Indv "a")) (nonInf (UnR myR (Indv "b")))

form15 :: Form
form15 = Forall "x" (nonInf (UnR myR (Var "x")))

form16 :: Form
form16 = Exists "x" (nonInf (UnR myR (Var "x")))
```

5 Simple Tests

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

```haskell
module Main where

import InqBModels
import InqBSyntax
import InqBSemantics
import ModelChecker
import Examples
import HelperFunctions ( powerset )
import Data.List
import Test.QuickCheck
import Test.Hspec

main :: IO()
main = hspec $ do
    describe "Basics" $ do
        it "funnyfunction: result is within [1..100]" $
            property trivialModelTest

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 = sort (toProp m f) == sort (toProp m g)

trivialTest :: Form -> Bool
trivialTest _ = True

trivialModelTest :: Model -> Bool
trivialModelTest _ = True

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

6 Conclusion

[Knu11]

References

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