My Report

Bo Flachs & Wessel Kroon

Sunday 30th January, 2022

Abstract

Contents

1 Introduction

2 Inquisitive Semantics

3 InqB in Haskell

3.1 Models

In this subsection we discuss the implementation of the *InqB* models as defined in Definition **??**. We make possible worlds of the type `Int` and individuals of the type `String`.

```
module InqBModels where
import Data.Functor.Contravariant (defaultEquivalence)

type World        = Int
type Universe     = [World]
type Individual   = String
type Domain       = [Individual]
```

Inquisitive semantics is designed so that relations can be n-ary for any $n \in \mathbb{N}$. However, in natural language we rarely encountered relations of an arity higher than three. We have therefore chosen to only implement unary, binary and tertiary relations. For example, the unary relation is represented as the characteristic set of a function from worlds to sets of individuals. [1]

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

Our models then consists of a universe, a domain, and lists of unary, binary and tertiary relations. Note that we diverge from Definition **??** in this respect. We omit the interpretation function I and replace this in two ways.

First, as the domain should be constant in all worlds, we work with the domain of the model rather than with a domain relative to a world.

Second, we do not work with relation symbols that are interpreted in a model. Instead we add the relations directly to the model. As we shall see shortly, this allows for a very straightforward way of defining models. The downside is that we do not have a fixed language with relation symbols that are interpreted differently in different models. This means that a formula is always defined relative to a model, as we will see in Section 3.2. We have chosen to put this restriction on our models so that the implementation of arbitrary models can be simpler. And although this might be mathematically less complete, it allows for an intuitive way of defining one's one models.

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

[1] Note that we have chosen to represent sets as lists in Haskell.

An example of an *InqB* model in this framework would then be as follows.

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

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

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

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

myModel2 :: Model
myModel2 = Mo [1, 2, 3, 4] ["a", "b"] [myR, myR2] [myBiR] [myTertR]
```

Lastly, we define information states and propositions as sets of worlds and sets of sets of worlds respectively.

```
type Prop     = [[World]]
type InfState = [World]
```

Given these implementations of an *InqB* model we can now implement the syntax of inquisitive semantics.

3.2 Syntax

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

```
module InqBSyntax where

import HelperFunctions
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
```

```haskell
newtype ModelWithForm = MWF (Model, Form) deriving Show

instance Arbitrary ModelWithForm where
    arbitrary = do
      u <- suchThat (sublistOf myWorlds) (not . null)
      d <- suchThat (sublistOf myIndividuals) (not . null)
      ur <- replicate 1 <$> (zip u <$> (sublistOf ((concat . replicate (length
           u) . powerset) d) >>= shuffle ))
      br <- replicate 1 <$> (zip u <$> sublistOf ((concat . replicate (length
          u) . powerset)
                  [(x,y)| x<-d,y<-d]))
      tr <- replicate 1 <$> (zip u <$> sublistOf ((concat . replicate (length
          u) . powerset)
              [(x,y,z)| x<-d, y<-d, z<-d]))
      let model = Mo u d ur br tr
      form <- sized (randomForm model)
      return (MWF (model, form)) where
        randomForm :: Model -> Int -> Gen Form
        randomForm m 0 = UnR <$> elements (unRel m)
                               <*> elements (map Indv (dom m))
        randomForm m n = oneof
            [ UnR   <$> elements (unRel m)
                    <*> elements (map Indv (dom m))
            , BinR  <$> elements (biRel m)
                    <*> elements (map Indv (dom m))
                    <*> elements (map Indv (dom m))
            , TertR <$> elements (tertRel m)
                    <*> elements (map Indv (dom m))
                    <*> elements (map Indv (dom m))
                    <*> elements (map Indv (dom m))
            , Neg   <$> randomForm m (n `div` 4)
            , Con   <$> randomForm m (n `div` 4)
                    <*> randomForm m (n `div` 4)
            , Dis   <$> randomForm m (n `div` 4)
                    <*> randomForm m (n `div` 4)
            , Impl  <$> randomForm m (n `div` 4)
                    <*> randomForm m (n `div` 4)
            ]
```

3.3 Semantics

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

```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
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 Model Checker

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

```
module ModelChecker where

import InqBModels
import InqBSyntax
import InqBSemantics

-- 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
```

3.5 Helper functions

In this subsection we discuss some helper functions that we implemented

```
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
```

4 Simple Tests

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

```
module Main where

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

main :: IO()
main = hspec $ do
    describe "Fact 4.12" $ do
        it "!phi equiv neg neg phi" $
            property (\(MWF (m, f))-> isEquivalent m (nonInq f) (Neg (Neg f))
                )
        it "?phi equiv phi or (neg phi)" $
            property (\(MWF (m, f))-> isEquivalent m (nonInf f) (Dis f $ Neg f
                ) )
    describe "Fact 4.13" $ do
        it "phi equiv (!phi and ?phi)" $
            property (\(MWF (m, f))-> isEquivalent m f (Con (nonInq f) (nonInf
                 f)) )
    describe "Fact 4" $ do
        it "2. (neg phi) is always non-inquisitive" $
            property (\(MWF (m, f))-> (not . isInquisitive m) (Neg f) )
        it "3. !phi is always non-inquisitive" $
            property (\(MWF (m, f))-> (not . isInquisitive m) (nonInq f) )
    describe "Fact 4.18" $ do
        it "1. ?phi is always non-informative" $
            property (\(MWF (m, f))-> (not . isInformative m) (nonInf f) )

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)
```

5 Conclusion

[Knu11]

References

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