My Report

Bo Flachs & Wessel Kroon

Monday 31$^{\text{st}}$ January, 2022

Abstract

abstract here

Contents

1 Introduction

Inquisitive semantics is a relatively new framework in which information exchange can be analysed. In addition to declarative sentences, questions can be analysed in this semantic framework. In this report we describe how we build a model checker for the most basic version of inquisitive semantics, InqB.

We have two goals in mind. Firstly, we want to be able to evaluate formulas relative to specified models. And secondly, we want to check several theorems using QuickCheck.

In Section 2 we will give a concise introduction to InqB, taking our cue from [CGR19]. Section 3 concerns itself with the implementation of InqB and our model checker in Haskell. Thereafter, in Section 4, we give some examples. In Section 5 we perform some simple tests after which we give our conclusion in Section 6.

2 Inquisitive Semantics

Any standard first-order language $\mathcal{L}$, containing a set of function symbols $\mathcal{F}_{\mathcal{L}}$ and a set of relation symbols $\mathcal{R}_{\mathcal{L}}$, is also a language of InqB. In our model checker we do not concern ourselves with function symbols, therefore we will not mention them in the remainder of this report. As for the constants in a language, we will assume that for each individual in the domain of a model we will have a constant in our language. We define models of InqB below.

Definition 1. *An InqB model for a first-order language $\mathcal{L}$ is a triple $M = \langle W, D, I\rangle$, where:*

- *W is a non-empty set of possible worlds;*
- *D is a non-empty set of individuals;*
- *I is a map that associates every $w \in W$ with a first order structure I_w such that:*
  - *for every $w \in W$, the domain of I_w is D;*
  - *for every n-ary relation symbol $R \in \mathcal{R}_{\mathcal{L}}$, $I_w(R) \subseteq D^n$;*

Before giving the semantics of InqB, we introduce some terminology. Instead of worlds, inquisitive semantics takes sets of worlds as primitive. A set of worlds is called an information state. A proposition, then, consists of a set of sets of worlds instead of a set of worlds as in classical logic.

Definition 2. *Let $M = \langle W, D, I\rangle$ be a model. An information state s is a set of possible worlds $s \subseteq W$. A proposition P is a non-empty, downwards closed set of information states. The proposition P corresponding to a formula φ (in a certain model) is denoted by $[\varphi]$.*

The information states in a proposition correspond to the states in which the issue raised by a proposition is resolved. As smaller information states provide *more* information, the requirement that propositions are downwards closed make sense.

Given Definition 2, we can characterize a proposition in terms of its maximal elements. We call these maximal elements the alternatives of a proposition. If a proposition has more than one alternative, we say that this proposition is inquisitive. Moreover, the union of the sets in a proposition can be viewed as the informative content of a proposition. We formally define these two notions, they will play a role in later sections.

Definition 3. *The alternatives of a proposition P, denoted by $alt(P)$, are its maximal elements. The informative content of a proposition P, denoted by $info(P)$, is defined as $info(P) := \cup P$.*

In classical logic, logical operations correspond to certain algebraic operations. This is also the case in inquisitive semantics. We will use the algebraic characterization of InqB's semantics in the implementation of our model checker, hence we also present the semantics in this way here.[1] In addition to union and intersection, two more algebraic operations are used: relative and absolute pseudo-complement.

Definition 4. *For propositions P and Q, the pseudo-complement of P relative to Q, denoted by $P \Rightarrow Q$, is defined as $P \Rightarrow Q := \{s \mid \text{for every } r \subseteq s, \text{ if } t \in P, \text{ then } t \in Q\}$. For any proposition P, the absolute pseudo-complement P^* of P is defined as $P^* := \{s \mid s \cap t = \emptyset \text{ for all } t \in P\}$.*

Using these algebraic operations, we can now give the semantics of InqB.

Definition 5. *The semantics of InqB are given by:*

1. $[R(t_1, \ldots, t_n)] := \mathcal{P}(|R(t_1, \ldots, t_n)|)$;
2. $[\neg\varphi] := [\varphi]^*$;
3. $[\varphi \wedge \psi] := [\varphi] \cap [\psi]$;
4. $[\varphi \vee \psi] := [\varphi] \cup [\psi]$;
5. $[\varphi \rightarrow \psi] := [\varphi] \Rightarrow [\psi]$;
6. $[\forall x.\varphi(x)] := \cap_{d \in D}[\varphi(d)]$;
7. $[\exists x.\varphi(x)] := \cup_{d \in D}[\varphi(d)]$,

where $|R(t_1, \ldots, t_n)|$ denotes the set of worlds where $R(t_1, \ldots, t_n)$ is classically true.

Lastly, we introduce two new projection operators: '!' and '?'. We call these projection operators because they project a proposition on the set of purely informative propositions and the set of purely inquisitive propositions, respectively.

Definition 6. *For any proposition P, we have:*

- $!P := \mathcal{P}(info(P))$;
- $?P := P \cup P^*$.

[1] For the semantics of InqB in terms of support conditions, see [CGR19, p. 62-63].

We can express $!P$ in terms of our algebraic operators as $!P = P^{**}$. It then follows from Definition 5 and Definition 6 that we can express the projection operators in terms of negation and disjunction: $!\varphi \equiv \neg\neg\varphi$ and $?\varphi \equiv \varphi \vee \neg\varphi$. So we do not have to add these operators to our language in order to make our language more expressive.

This concludes our introduction to InqB. In the next section we will discuss how we implemented InqB in Haskell.

3 InqB in Haskell

3.1 Models

We now discuss the implementation of the *InqB* models as defined in Definition 1. We make possible worlds of the type `Int` and individuals of the type `String`.

```
module InqBModels where

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. [2]

```
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 1 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]
```

[2] Note that we represent sets as lists in Haskell.

```
                 , biRel :: [BiRelation]
                 , tertRel :: [TertRelation] }
        deriving (Eq, Ord, Show)
```

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

We now discuss the implementation of the syntax of *InqB*. We say that a variable is of the type `String`, and a term is either an individual or a variable.

```
module InqBSyntax where

import HelperFunctions
import InqBModels
import Test.QuickCheck

type Var  = String
data Term = Indv Individual | Var Var
        deriving (Eq, Ord, Show)
```

We can then define formulas in a way that is analogous to the Backus-Naur form for first-order formulas. Note that we do not add a relation symbol in the atomic sentences, but the actual relation. As discussed in Section 3.1, this may seem like an unnatural way to define formulas. However, as we we will see shortly, it is still intuitive to define one's own formulas in this way. Furthermore, it allows for a straightforward implementation of arbitrary models and 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)
```

The projection operators ! (`nonInq`) and ? (`nonInf`) can be seen as abbreviations. Therefore we have implemented them as functions of the type `Form` → `Form`.

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

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

We can then define an example formula using the example relation `myR` from Section 3.1. This formula corresponds to the *InqB* formula $!(Ra \vee Rb)$.

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

Now that we have defined what an *InqB* model and an *InqB* formula looks like in Haskell, we can create arbitrary instances of them. We do this by creating a new type that is a tuple of a model and a formula. Thereby we can create a single instance of the class `Arbitrary`, which we can use for the checking of several *InqB* facts. These checks are implemented using QuickCheck and are discussed in Section 4.

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

For an arbitrary `ModelWithForm` we fix a set of world and a set of individuals. The model then will contain an arbitrary subset of these. As the `Arbitrary` instance is quite long, we will go over the code line by line.

First we let the universe, `u`, and the domain, `d`, be arbitrary non-empty subsets of `myWorlds` and `myIndividuals`, respectively. We then take an arbitrary and shuffled list of lists of individuals and zip this with the universe. Replicating this gives us something of the type `UnRelation`. We have chosen to only include one relation of each arity in our arbitrary models, but this could be extended to an arbitrary number. The code for binary, `ur`, and tertiary, `tr`, relations is analogous.[3] Putting these all together we have an arbitrary model.

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

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

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
```

[3]Note that these sets are not shuffled. This is because the shuffling of of these larger sets made the code very slow. A possible improvement would be a way of getting arbitrary relations with less overhead.

Using the function `sized` we can then create formulas of arbitrary length using the individuals and relations that were created for this arbitrary model. We have not implemented arbitrary formulas containing quantifiers, as this poses a rather difficult extra challenge. To correctly implement this one could first create a formula, and then substitute some of the individuals for the variable that will be quantified over. However, for our current purposes and time constraints we have chosen to work with these restrictions.

```
      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)]
```

Now that we have defined what models and formulas are, we can implement the semantics of *InqB*.

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
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
```

We should not forget to give an example with a nice tikz picture here!!!! ldsjlaflsajf

3.4 Model Checker

Now that we have implemented the models, syntax and semantics of InqB, our model checker can me implemented. As a result of our algebraic characterization of InqB's semantics, the support of a proposition in an information state comes down to set inclusion. The function `supportsProp` takes an information state and a proposition and checks whether that information state is included in the proposition. If so, the information state supports the propositions, otherwise it does not.

```
module ModelChecker where

import InqBModels
import InqBSyntax
import InqBSemantics

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

Likewise, the function supportsForm checks whether an information state supports a certain formula. However, this can only be checked by transforming the formula into a proposition relative to a certain model. Hence supportsForm also takes a Model as an argument.

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

Lastly, the function makesTrue checks whether a formula is satisfied in a certain world of a particular model. This comes down to checking whether the singleton containing that particular world is an element of the proposition corresponding to the specified formula. Note that checking whether a formula is true in some world comes down to checking whether that formula is supported in the information state containing only that world.

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

3.5 Helper functions

We conclude this section with three helper functions that we used in the implementations above. The functions below can be used to generate the power set of a list, check whether something is a strict subset and to give the downward closure of a set of sets, respectively. These functions are used in the code treated in sections 3.2 and 3.3. Finally, note that the closeDownward function can be used to transform the alternatives of a proposition, $\text{alt}(P)$, into the proposition P.

```
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 QuickCheck

Now that we have defined *InqB* models, formulas, propositions and a model checker, we can use QuickCheck to check several facts about *InqB*.[4] These facts are from [CGR19], and we use their numbering.

We will first list these facts, after which we will discuss the QuickCheck implementation.

[4]We say facts rather than propositions or theorems, because this is the terminology that is used in the original sources.

- **Fact 4.12**
  - $!\varphi \equiv \neg\neg\varphi$
  - $?\varphi \equiv \varphi \vee \neg\varphi$
- **Fact 4.13**: $\varphi \equiv (!\varphi \wedge ?\varphi)$
- **Fact 4.17**
  - *2.*: $\neg\varphi$ is always non-inquisitive.
  - *3.*: $!\varphi$ is always non-inquisitive.
- **Fact 4.18**
  - *1.*: $?\varphi$ is always non-informative.

For the implementation of the tests of these facts, we import all modules that were discussed up until now.

```
module Main where

import InqBModels
import InqBSyntax
import InqBSemantics
import HelperFunctions
import Data.List
import Test.QuickCheck
import Test.Hspec
```

The `main` function implements all of the facts listed above as properties. We use Hspec, QuickCheck and the `Arbitrary` instance of `ModelWithForm` to test these facts.

```
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 4.17" $ 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))
```

For these properties we have used three functions that implement when a formulas are inquisitive, informative, and equivalent. These correspond to the following three definitions:

- A formula is informative iff `info`$(\varphi) \neq W$;
- A formula is inquisitive iff `info`$\varphi \notin [\varphi]$;
- $\varphi \equiv \psi$ just in case that $[\varphi] = [\psi]$.

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

isInquisitive :: Model -> Form -> Bool
isInquisitive m f = sort (toProp m f) /= (sort . powerset) (info m f)

isEquivalent :: Model -> Form -> Form -> Bool
isEquivalent m f g = sort (toProp m f) == sort (toProp m g)
```

These tests can be run by using the command `stack test`. This will give the following output:

```
imc> test (suite: tests)

Fact 4.12
  !phi equiv neg neg phi
    +++ OK, passed 100 tests.
  ?phi equiv phi or (neg phi)
    +++ OK, passed 100 tests.
Fact 4.13
  phi equiv (!phi and ?phi)
    +++ OK, passed 100 tests.
Fact 4.17
  2. (neg phi) is always non-inquisitive
    +++ OK, passed 100 tests.
  3. !phi is always non-inquisitive
    +++ OK, passed 100 tests.
Fact 4.18
  1. ?phi is always non-informative
    +++ OK, passed 100 tests.

Finished in 0.0088 seconds
6 examples, 0 failures

imc> Test suite tests passed
```

We can therefore conclude that our implementation of *InqB* in Haskell works correctly. Furthermore, we have implemented a tool that can be used to check more involved facts about the framework *InqB*.

5 Conclusion

References

[CGR19] Ivano Ciardelli, Jeroen Groenendijk, and Floris Roelofsen. *Inquisitive semantics*. Oxford University Press, 2019.