A Model Checker for Inquisitive Semantics

Bo Flachs & Wessel Kroon

Thursday 3rd February, 2022

Abstract

In this report we discuss a Haskell model checker for basic inquisitive semantics, called InqB. Inquisitive semantics is designed to enable us to analyse not only declarative natural language sentences, but also interrogatives. We give a short introduction into the technicalities of the framework, after which we discuss its implementation in Haskell step by step. We use QuickCheck to check several well-known facts about the framework InqB.

Contents

1 Introduction

Inquisitive semantics is a logical framework in which information exchange in natural language can be analysed. In addition to declarative sentences, questions can also be analysed in this semantic framework. In this report we describe the implementation of 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 facts using QuickCheck.

In Section 2 we will give a concise introduction to InqB. Section 3 concerns itself with the implementation of InqB and our model checker in Haskell. In Section 4 we check several well-known facts of InqB, after which we give our conclusion in Section 5.

2 Inquisitive Semantics

Below we will give a brief overview of InqB, taking our cue from [CGR19].[1]

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*

[1] Basic notions such as propositions, information states and alternatives are treated comprehensively in Chapter 2 of [CGR19], while the language, models and semantics of InqB are treated in Chapter 4 of the same book.

information states. The proposition P corresponding to a formula φ (in a certain model) is denoted by $[\varphi]$.

Given Definition 2, we can characterize a proposition in terms of its maximal elements, which we will call alternatives.

Definition 3. *The alternatives of a proposition P, denoted by $\mathtt{alt}(P)$, are its maximal elements. If $|\mathtt{alt}(P)| \neq 1$, we say that P is inquisitive, otherwise P is non-inquisitive.*

The issue raised by a proposition, then, can intuitively be understood as the problem of not knowing which of its alternatives is the case. The information states in a proposition correspond to the states in which the issue raised by a proposition is resolved, i.e. the states in which one is able to know for at least one alternative that it is the case. As smaller information states provide *more* information, the requirement that propositions are downwards closed make sense. A proposition P can be retrieved from its alternatives $\mathtt{alt}(P)$ by taking the downwards closure of $\mathtt{alt}(P)$.

If a proposition P has more than one alternative, it expresses the question which of its alternatives is the case. If there is only one alternative, then the proposition simply expresses the information contained in that single alternative, hence it makes sense that we call such a proposition non-inquisitive. Note that a proposition cannot have 0 alternatives by definition.[2]

Besides being inquisitive or not, propositions can be informative or non-informative. The union of the sets in a proposition can be viewed as the informative content of a proposition. If the informative content of a proposition does not contain all worlds, then we say that the proposition is informative: it provides us with the information that the excluded worlds are not the case.

Definition 4. *The informative content of a proposition P is defined as $\mathtt{info}(P) := \cup P$. A proposition P is informative iff $\mathtt{info}(P) \neq W$ (where W is the set of worlds in a model).*

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.[3] In addition to union and intersection, two more algebraic operations are used: relative and absolute pseudo-complement.

Definition 5. *For propositions P and Q, the pseudo-complement of P relative to 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 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.

[2]For infinite models it is possible to have 0 alternatives, see footnote 3 of [CGR19, p. 20]. However, we are only concerned with finite models in this report.

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

Definition 6. *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. $[\varphi \to \psi] := [\varphi] \Rightarrow [\psi]$;
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.*

The definition above gives us a semantics in the sense that every formula φ is associated with a proposition $[\varphi]$, and that we can say that φ is supported by an information state s whenever $s \in [\varphi]$. If one prefers to speak about satisfaction in worlds, we can say that φ is satisfied at a world w if $\{w\} \in [\varphi]$.

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

Definition 7. *For any proposition P, we have:*

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

We can express $!P$ in terms of our algebraic operators as $!P = P^{**}$. It then follows from Definition 6 and Definition 7 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, as they can already be seen as abbreviations.

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`. This allows us to implement a universe and a domain in a principled manner.

```haskell
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 encounter 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.[4]

```
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 models.

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

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

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

myUnR2 :: UnRelation
myUnR2 = [(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")])]

myModel :: Model
myModel = Mo [1, 2, 3, 4] ["a", "b"] [myUnR, myUnR2] [myBiR] [myTertR]
```

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

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

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

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.

```haskell
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.

```haskell
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 $\rightarrow$ Form.

```haskell
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 myUnR from Section 3.1. This formula corresponds to the InqB formula $?!(Ra \vee Rb)$.

```haskell
myForm :: Form
myForm = (nonInf . nonInq) (Dis (UnR myUnR (Indv "a")) (UnR myUnR (Indv "b")))
```

Now that we have defined what an InqB model and an InqB formula look 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.

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

For an arbitrary `ModelWithForm` we fix a set of world and a set of individuals. The arbitrary model 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`, called `ur`. 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.[5] 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
```

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, given 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)
```

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

```
            , 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

We can implement the semantics of InqB using the models and syntax discussed in previous sections. Firstly, we need to implement the algebraic operators that we do not yet have in Haskell: the relative and absolute pseudo-complement. We follow Definition 5.

```
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
```

In order to work with quantifiers and variables, we need to be able to substitute elements from our domain. The following function, `substitute`, substitutes an individual in place of a variable in a formula, yielding a new formula.

```
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
```

The next step consists of writing a function that enables us to turn a formula into a proposition relative to some model. We will see that in order to do this we need to be able to convert objects of type `Term` into objects of type `String`. The function `getString` does this. We have chosen to not put this function in the module with helper functions, to avoid cyclic importations.

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

Now, the function `toProp` turns formulas into propositions relative to a model. The function is defined recursively and mirrors Definition 6. Observe that the cases for atomic formulas do not use the model. Instead, the relations are used in these clauses, because the required information for constructing a proposition is already present in the relation `r`, which is part of a model. This is a consequence of not using relation symbols in formulas but the relations themselves. This is a shortcoming that we already discussed in Section 3.1. The clauses for non-atomic formulas are straightforward implementations of Definition 6.

```
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 ]
```

There are two additional functions part of our semantics. Firstly, we have the function `alt`, turning a formula into the set of alternatives of its corresponding formula, relative to a model. This is done using `toProp` and then taking the maximal elements of the resulting set of information states, in accordance with Definition 3. We sort the resulting set of information states to avoid having equivalent propositions that are not recognized as such by Haskell.

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

Secondly, we have the function `info`, giving the informative content of a formula relative to a model. This function takes the union of all information states in a proposition, in accordance with Definition 4. The resulting information state is sorted for the same reason as in the function above.

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

We consider an example. In Figure 1 we have given a visual representation of an *InqB* model M. The grey areas correspond to the two alternatives, i.e., maximal elements, of the proposition corresponding to the formula $?!(Ra \vee Rb)$. Note that this formula is inquisitive as there are multiple alternatives. However, as the alternatives together cover the whole universe, we see that it is not informative.

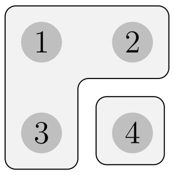

Figure 1: The model M with the proposition $[?!(Ra \vee Rb)]$.

In Section 3.1 we have seen how we can represent this model in Haskell, i.e. M corresponds to `myModel`. Similarly, we have already seen the implementation of $?!(Ra \vee Rb)$ as `myForm`. Lastly, we can compute the alternatives of the proposition $[?!(Ra \vee Rb)$ in GHCI as follows:

```
*InqBSemantics> alt myModel myForm
```

This will return the following list of lists, which corresponds exactly to the alternatives in the figure.

```
[[1,2,3],[4]]
```

3.4 Model Checker

Now that we have implemented the models, syntax and semantics of InqB, our model checker can be 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 this is just checking whether that formula is supported by the information state containing only that world.

```
makesTrue :: Model -> World -> Form -> Bool
makesTrue m w = supportsForm m [w]
```

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, 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.[6] These facts are from [CGR19], and we use their numbering. We will first list these facts, after which we will discuss the QuickCheck implementation.

- **Fact 4.12**
  - $!\varphi \equiv \neg\neg\varphi$
  - $?\varphi \equiv \varphi \vee \neg\varphi$
- **Fact 4.13**: $\varphi \equiv (!\varphi \wedge ?\varphi)$

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

- **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 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 $[\varphi] = [\psi]$.

```
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

In this report we gave a concise introduction to the most basic framework of inquisitive semantics (InqB). We then implemented the models, syntax and semantics of InqB in Haskell. Thereafter, we combined these to create a model checker. Lastly, we used QuickCheck to check several facts about the framework InqB. Thereby we have created a tool that can be used to evaluate more complex models and formulas, and to check complex facts about InqB.

However, our implementation of InqB has two shortcomings. Firstly, as discussed in Section 3.1, we have omitted an interpretation function. This allowed us to give a simpler, more explicit manner of defining an arbitrary model. However, the downside of this is that formulas do not contain relation symbols but actual relations. Consequently, we can only define a formula relative to a model, making it impossible to evaluate the same formula in several different models.

Secondly, our `Arbitrary` instance does not allow for arbitrary formulas containing quantifiers. This does not make our system less expressive, as quantifiers can be expressed using conjunction and disjunction in finite models. However, it would be an improvement to allow for the more succinct representation of formulas using quantifiers.

Our implementation can be improved by overcoming these two shortcomings. In sections 3.1 and 3.2 we have discussed possible approaches for these improvements. Our model checker could also be extended to implement Inquisitive Epistemic

Logic (IEL) which is a proper extension of InqB. Furthermore, an implementation for inquisitive semantics using intuitionistic logic rather than classical logic (InqI), would be an interesting subject for future research.

References

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