Model Checking the Hardest Logic Puzzle Ever

Malvin Gattinger
malvin@w4eg.eu

Last Update: Tuesday 3rd May, 2016

Abstract

We implement the logics of Agent Types from [LW13] in Haskell. The resulting program checks various Theorems in fractions of seconds. In particular we verify solutions to the so-called Hardest Logic Puzzle Ever.

The implementation is stand-alone, while similar to DEMO-S5 [vE07]. We make heavy use of Haskell's pattern matching and models are represented explicitly, using a custom data type based on lists.

Contents

1 Agent-Type Logic

```
module AGTYPES where

type Prop = Int
type AgentName = String
```

An agent type is a function which takes an agent name and a formula as arguments and returns a formula. The returned formula is the precondition for an agent to announce the given formula. Additionally our data type `AgentType` contains a string to allow equality checks.

```
data AgentType = AgT String (AgentName -> Form -> Form)
instance (Eq AgentType) where (==) (AgT s _) (AgT t _) = s == t
instance (Show AgentType) where show (AgT s _) = s

truthTeller, liar, bluffer, subjTruthTeller, subjLiar :: AgentType
truthTeller     = AgT "truthTeller" (const id)
liar            = AgT "liar"        (const Neg)
bluffer         = AgT "bluffer"     (const $ const Top)
subjTruthTeller = AgT "subjTruthTeller" K
subjLiar        = AgT "subjLiar" (\ i f -> Neg $ K i f)
```

1.1 Syntax and Semantics

Definition 2: Language

```
data Form =
  Top | Prp Prop | AgentName `IsA` AgentType
  | Neg Form | Conj [Form]
  | K AgentName Form
  | Announce AgentName Form Form
  | GodAnnounce Form Form
  deriving (Eq,Show)
```

Abbreviations:

```
disj :: [Form] -> Form
disj fs = Neg $ Conj (map Neg fs)
impl :: Form -> Form -> Form
impl f g = disj [Neg f, g]

kw :: AgentName -> Form -> Form
kw a f = disj [K a f, K a (Neg f)]

diamAnnounce :: AgentName -> Form -> Form -> Form
diamAnnounce a f g = Neg $ Announce a f (Neg g)
```

Definition 3: $\texttt{PALT}^{\texttt{T}}$ Models and Semantics

```
type World = Int
type Partition a = [[a]]
type Label = [Prop]
type Model = ( [(World, (Label, [(AgentName, AgentType)]))],
               [(AgentName, Partition World)] )
type Scene = (Model, World)

labelAt :: Scene -> Label
labelAt ((ws,_),w) = l where Just (l,_) = lookup w ws

typeAtOf :: Scene -> AgentName -> AgentType
typeAtOf ((ws,_),w) a = t where
  Just (_,tOf) = lookup w ws
  Just t = lookup a tOf

partAtOf :: Scene -> AgentName -> [World]
partAtOf ((_,pss),w) i = head $ filter (elem w) ps
  where Just ps = lookup i pss
```

```
implies :: Bool -> Bool -> Bool
implies a b = not a || b

isTrue :: Scene -> Form -> Bool
isTrue _  Top          = True
isTrue sc (Prp p)      = p `elem` labelAt sc
isTrue sc (a `IsA` t) = typeAtOf sc a == t
isTrue sc (Neg f)      = not $ isTrue sc f
isTrue sc (Conj fs)    = all (isTrue sc) fs
isTrue (m,w) (K a f)   = all (\v -> isTrue (m,v) f) (partAtOf (m,w) a)
isTrue sc@(m,w) (Announce a f g) =
  canSay sc a f `implies` isTrue (announce m (Just a) f, w) g
isTrue sc@(m,w) (GodAnnounce f g) =
  isTrue sc f `implies` isTrue (announce m Nothing f, w) g

canSay :: Scene -> AgentName -> Form -> Bool
canSay sc a f = isTrue sc (agtf a f) where AgT _ agtf = typeAtOf sc a
```

Announcement: Given a model, let agent a announce f. If no a is given, let god announce it.

```
announce :: Model -> Maybe AgentName -> Form -> Model
announce oldm@(oldws,oldpss) ma f = (ws,pss) where
  ws = filter check oldws
  check (v,_) = case ma of
    Nothing -> isTrue (oldm,v) f
    Just a -> isTrue (oldm,v) (agtf a f) where AgT _ agtf = typeAtOf (oldm,v) a
  pss = map (fmap strip) oldpss
  strip set = filter ([]/=) $ map (filter (`elem` map fst ws)) set
```

Validity is truth at all worlds:

```
isValid :: Model -> Form -> Bool
isValid m f = and [ isTrue (m,w) f | w <- map fst (fst m) ]
```

A toy example:

```
myM :: Model
myM = ( [ (0,([0],[("Alice",liar)])),
          (1,([1],[("Alice",liar)])) ],
        [ ("Alice", [[0],[1]]) ] )
```

```
*AGTYPES> canSay (myM,0) "Alice" (Prp 0)
False
*AGTYPES> canSay (myM,0) "Alice" (Prp 1)
True
*AGTYPES> canSay (myM,1) "Alice" (Prp 1)
False
*AGTYPES> canSay (myM,1) "Alice" (Prp 0)
True
```

1.2 Example 3: Three inhabitants

```
example3m1, example3m2, example3m3, example3m4 :: Model
example3m1 =
  ( [ (1,([],[("A",liar),("B",liar),("C",liar)]))
    , (2,([],[("A",liar),("B",liar),("C",truthTeller)]))
    , (3,([],[("A",liar),("B",truthTeller),("C",truthTeller)]))
    , (4,([],[("A",liar),("B",truthTeller),("C",liar)]))
    , (5,([],[("A",truthTeller),("B",truthTeller),("C",truthTeller)]))
    , (6,([],[("A",truthTeller),("B",truthTeller),("C",liar)]))
    , (7,([],[("A",truthTeller),("B",liar),("C",liar)]))
    , (8,([],[("A",truthTeller),("B",liar),("C",truthTeller)]))
    ], [ ("D", [[1..8]]) ] )

example3m2 = announce example3m1 (Just "A") ("B" `IsA` liar)
example3m3 = announce example3m2 (Just "B") ("C" `IsA` liar)
example3m4 = announce example3m3 (Just "C") (Conj ["A" `IsA` liar, "B" `IsA` liar])
```

```
*AGTYPES> example3m4
([(4,([],[("A",liar),("B",truthTeller),("C",liar)]))],[("D",[[4]])])
```

1.3 Example 4: Death or Freedom

```
example4m1 :: Model
example4m1 =
  ( [ (1,([1],[("A",truthTeller),("B",liar)]))
    , (2,([1],[("A",liar),("B",truthTeller)]))
    , (3,([ ],[("A",truthTeller),("B",liar)]))
    , (4,([ ],[("A",liar),("B",truthTeller)]))
    ], [ ("C", [[1..4]]) ] )
```

"If I am a Knight, then the road behind me leads to Freedom, and if I am a Knave, then the road behind me leads to Death.":

```
example4solution :: Form
example4solution = Conj
  [ impl ("A" `IsA` truthTeller) (Prp 1)
  , impl ("A" `IsA` liar) (Neg $ Prp 1) ]
```

```
*AGTYPES> isTrue (example4m1,1) (diamAnnounce "A" example4solution $ K "C" (Prp 1))
True
```

```
example4solutionAlt :: Form
example4solutionAlt =
  ("A" `IsA` truthTeller) `impl` Conj
    [ Prp 1          `impl` diamAnnounce "A" (diamAnnounce "B" (Neg $ Prp 1) Top) (kw "C" (
        Prp 1))
    , (Neg $ Prp 1) `impl` diamAnnounce "A" (diamAnnounce "B" (Neg $ Prp 1) Top) (kw "C" (
        Prp 1)) ]
```

```
*AGTYPES> isTrue (example4m1,1) example4solutionAlt
True
```

Making the example harder:

```
example4harderm1 :: Model
example4harderm1 =
  ( [ (1,([1],[("A",subjTruthTeller),("B",subjTruthTeller)]))
    , (2,([1],[("A",subjTruthTeller),("B",subjLiar)]))
    , (3,([ ],[("A",subjTruthTeller),("B",subjLiar)]))
    , (4,([ ],[("A",subjTruthTeller),("B",subjTruthTeller)]))
    , (5,([1],[("A",subjLiar),("B",subjTruthTeller)]))
    , (6,([1],[("A",subjLiar),("B",subjLiar)]))
    , (7,([ ],[("A",subjLiar),("B",subjLiar)]))
    , (8,([ ],[("A",subjLiar),("B",subjTruthTeller)]))
    ], [ ("A",[[1,2],[3,4],[5,6],[7,8]])
       , ("B",[[1,5],[2,6],[3,7],[4,8]])
       , ("C", [[1..8]]) ] )

example4harderValid :: Form
example4harderValid = Conj
  [ Neg $ kw "A" ("B" `IsA` subjTruthTeller)
  , K "C" $ Neg $ kw "A" ("B" `IsA` subjTruthTeller)
  , kw "A" (Prp 1)
  , Neg $ kw "C" (Prp 1)
  , Neg $ kw "C" ("A" `IsA` subjTruthTeller) ]
```

```
*AGTYPES> isValid example4harderm1 example4harderValid
True
```

We can check that the previous solution and a subjective variant do not work any more:

```
example4solutionAltSubj :: Form
example4solutionAltSubj =
  ("A" `IsA` subjTruthTeller) `impl` Conj
```

```
    [ Prp 1          `impl` diamAnnounce "A" (diamAnnounce "B" (Neg $ Prp 1) Top) (kw "C" (
        Prp 1))
    , (Neg $ Prp 1) `impl` diamAnnounce "A" (diamAnnounce "B" (Neg $ Prp 1) Top) (kw "C" (
        Prp 1)) ]
```

```
*AGTYPES> isTrue (example4harderm1,2) (diamAnnounce "A" (diamAnnounce "B" (Neg $ (Prp 1))
    Top) (kw "C" (Prp 1)))
False
*AGTYPES> isTrue (example4harderm1,2) example4solutionAltSubj
False
```

However, we can fix the short solution to "I would say my path leads to Freedom (if I were asked)".

```
check1, check2, modSolution :: Form
check1 = diamAnnounce "A" (diamAnnounce "A" (Prp 1) Top) Top
check2 = diamAnnounce "A" (Prp 1) Top
modSolution =
  ("A" `IsA` subjTruthTeller) `impl` Conj
    [ Prp 1          `impl` diamAnnounce "A" (diamAnnounce "A" (Prp 1) Top) (kw "C" (Prp 1))
    , (Neg $ Prp 1) `impl` diamAnnounce "A" (diamAnnounce "A" (Neg $ Prp 1) Top) (kw "C" (
        Prp 1)) ]
```

```
*AGTYPES> isTrue (example4harderm1,5) check1
True
*AGTYPES> isTrue (example4harderm1,5) check2
False
*AGTYPES> isValid example4harderm1 modSolution
True
```

1.4 Translation to PALT

We can translate PALTT to PALT as follows:

```
delta :: [AgentType] -> AgentName -> Form -> Form
delta allT a phi =
  disj [ Conj [ a `IsA` eta, agTfct a phi ] | eta@(AgT _ agTfct) <- allT ]

paltt2palt :: [AgentType] -> Form -> Form
paltt2palt _    Top = Top
paltt2palt _    (Prp p) = Prp p
paltt2palt _    (a `IsA` eta) = a `IsA` eta
paltt2palt allT (Neg f) = Neg $ paltt2palt allT f
paltt2palt allT (Conj fs) = Conj $ map (paltt2palt allT) fs
paltt2palt allT (K ag f) = K ag (paltt2palt allT f)
paltt2palt allT (Announce a psi phi) =
  GodAnnounce (paltt2palt allT $ delta allT a psi) (paltt2palt allT phi)
paltt2palt allT (GodAnnounce psi phi) =
  GodAnnounce (paltt2palt allT psi) (paltt2palt allT phi)
```

2 Question-Answer Logic

We now bring questions into the picture, i.e. into the language. In order not to go back and change stuff above we start mainly from scratch.

```
module QALogic where
```

There are some bits we can reuse though:

```
import AGTYPES (implies,AgentName,Prop,Partition,World,Label)
```

2.1 Syntax and Semantics

Language, Agent Types and Models have to be redefined.
Definition 4: Language PQLTT

```
data QForm =
  Top | Prp Prop | AgentName `IsA` AgentType
  | Neg QForm | Conj [QForm]
  | K AgentName QForm
  | Announce AgentName QForm QForm
  | Ask AgentName QForm QForm
  | Answer AgentName QForm
  deriving (Eq,Show)
```

Abbreviations:

```
disj :: [QForm] -> QForm
disj fs = Neg $ Conj (map Neg fs)

impl :: QForm -> QForm -> QForm
impl f g = disj [Neg f, g]

kw :: AgentName -> QForm -> QForm
kw a f = disj [K a f, K a (Neg f)]

-- diamonds
diam :: (QForm -> QForm) -> QForm -> QForm
diam act f = Neg $ act (Neg f)

diamAnnounce :: AgentName -> QForm -> QForm -> QForm
diamAnnounce a f = diam $ Announce a f
```

We also have to define agent types again because now they map different types of formulas.

```
data AgentType = AgT String (AgentName -> QForm -> QForm)
instance (Eq AgentType) where
  (==) (AgT s _) (AgT t _) = s == t
instance (Show AgentType) where
  show (AgT s _) = s

truthTeller, liar, bluffer, subjTruthTeller, subjLiar :: AgentType
truthTeller     = AgT "truthTeller" (const id)
liar            = AgT "liar"        (const Neg)
bluffer         = AgT "bluffer"     (const $ const Top)
subjTruthTeller = AgT "subjTruthTeller" K
subjLiar        = AgT "subjLiar" (\ i f -> Neg $ K i f)

labelAt :: Scene -> Label
labelAt ((ws,_),w) = l where Just (l,_) = lookup w ws

typeAtOf :: Scene -> AgentName -> AgentType
typeAtOf ((ws,_),w) a = t where
  Just (_,tOf) = lookup w ws
  Just t = lookup a tOf

partAtOf :: Scene -> AgentName -> [World]
partAtOf ((_,pss),w) i = head $ filter (elem w) ps where Just ps = lookup i pss

type Model = ( [(World, (Label, [(AgentName, AgentType)]))],
               [(AgentName, Partition World)] )
type Scene = (Model, World)
```

Definition 5: PQLTT Semantics

```
type Context = Maybe (AgentName,QForm)

qIsTrue :: Scene -> Context -> QForm -> Bool
qIsTrue _  _ Top          = True
qIsTrue sc _ (Prp p)      = p `elem` labelAt sc
qIsTrue sc _ (a `IsA` t) = typeAtOf sc a == t
qIsTrue sc c (Neg f)      = not (qIsTrue sc c f)
qIsTrue sc c (Conj fs)    = all (qIsTrue sc c) fs
qIsTrue (m,w) c (K a f)   = all (\v -> qIsTrue (m,v) c f) (partAtOf (m,w) a)
```

```
qIsTrue sc _ (Ask a f g) = qIsTrue sc (Just (a,f)) g

qIsTrue _ Nothing (Announce{}) = error "Nobody asked anything!"
qIsTrue sc@(m,w) c@(Just (a1,q)) (Announce a2 f g) =
  (a1 == a2) && f `elem` [q,Neg q] && qCanSay sc a1 f
    `implies` qIsTrue (qAnnounce m c (Just a1) f, w) Nothing g
```

Note that the semantic clause for answers quantifies over all formulas. This is not feasible in the implementation as there are infinitely many formulas. But we know that only the two formulas which count as an answer, i.e. the formula representing the question and its negation, are relevant. For all other formulas the Announce-Box will be trivially true because.

```
qIsTrue sc c (Answer a g) = all (\f -> qIsTrue sc c (Announce a f g)) relevantFormulas
    where
  relevantFormulas = case c of
    Nothing -> []
    Just (_,q) -> [q,Neg q]

qCanSay :: Scene -> AgentName -> QForm -> Bool
qCanSay sc a f = qIsTrue sc Nothing (agtf a f) where AgT _ agtf = typeAtOf sc a

qAnnounce :: Model -> Context -> Maybe AgentName -> QForm -> Model
qAnnounce oldm@(oldws,oldpss) c ma f = (ws,pss) where
  ws = filter check oldws
  check (v,_) = case ma of
    Nothing -> qIsTrue (oldm,v) c f
    Just a -> qIsTrue (oldm,v) Nothing (agtf a f) where AgT _ agtf = typeAtOf (oldm,v) a
  pss = map (fmap strip) oldpss
  strip set = filter ([]/=) $ map (filter (`elem` map fst ws)) set
```

As usual, validity is truth at all worlds:

```
qIsValid :: Model -> QForm -> Bool
qIsValid m f = and [ qIsTrue (m,w) Nothing f | w <- map fst (fst m) ]
```

2.2 Example 5: Death or Freedom with questions

```
example5m1 :: Model
example5m1 =
  ( [ (1,([1],[("A",truthTeller),("B",liar)]))
    , (2,([1],[("A",liar),("B",truthTeller)]))
    , (3,([ ],[("A",truthTeller),("B",liar)]))
    , (4,([ ],[("A",liar),("B",truthTeller)]))
    ], [ ("C", [[1..4]]) ] )

ex5q1, ex5q2 :: QForm -> QForm
-- Will the other man tell me that your path leads to Freedom?
ex5q1 = Ask "A" ( Ask "B" (Prp 1) (diamAnnounce "B" (Prp 1) Top) )
-- Will you say 'yes' if you are asked whether your path leads to Freedom?
ex5q2 = Ask "A" ( Ask "A" (Prp 1) (diamAnnounce "A" (Prp 1) Top) )
-- Both questions ensure that C learns whether Prp 1 holds.
ex5validity :: QForm
ex5validity = Conj [ action $ Answer "A" $ kw "C" (Prp 1) | action <- [ex5q1,ex5q2] ]
```

Result:

```
*QALogic> qIsValid example5m1 ex5validity
True
```

3 Question-Utterance Logic

Additionally to questions, we now also add utterances to our formalism.

```
module QULogic where
import Control.Arrow (second)
import Data.List (delete,groupBy,permutations,sort,sortOn)
import Data.Maybe
import HELP (fusion,(!))
import AGTYPES (implies,AgentName,Prop,Partition,World,Label)
```

Language, Agent Types and Models have to be redefined, again.

3.1 Syntax

Definition 6: Language PQLTTU

```
type Utterance = String

data QUForm =
  Top | Prp Prop | AgentName `IsA` AgentType
  | Neg QUForm | Conj [QUForm] | Disj [QUForm]
  | Impl QUForm QUForm
  | K AgentName QUForm
  | C [AgentName] QUForm
  | Say AgentName Utterance QUForm
  | Ask AgentName QUForm QUForm
  | Ans AgentName QUForm
  deriving (Eq,Show)
```

Note that we now also added a common knowledge operator `C` which is not present in [LW13]. We are not interested in the axiomatization of our logic here, hence it easy to deal with.

```
vee :: QUForm -> QUForm -> QUForm
vee f g = Disj [f,g]

kw :: AgentName -> QUForm -> QUForm
kw a f = Disj [K a f, K a (Neg f)]

diam :: (QUForm -> QUForm) -> QUForm -> QUForm
diam act f = Neg $ act (Neg f)

diamSay :: AgentName -> Utterance -> QUForm -> QUForm
diamSay a u = diam $ Say a u

diamAns :: AgentName -> QUForm -> QUForm
diamAns a = diam $ Ans a
```

We also have to define agent types again because now they map different types of formulas.

```
data AgentType = AgT String (AgentName -> QUForm -> QUForm)
instance (Eq AgentType) where
  (==) (AgT s _) (AgT t _) = s == t
instance (Show AgentType) where
  show (AgT s _) = s
instance (Ord AgentType) where
  compare (AgT s1 _) (AgT s2 _) = compare s1 s2

truthTeller, liar, bluffer, subjTruthTeller, subjLiar :: AgentType
truthTeller     = AgT "TT" (const id)
liar            = AgT "LL" (const Neg)
bluffer         = AgT "LT" (const $ const Top)
subjTruthTeller = AgT "STT" K
subjLiar        = AgT "SLL" (\ i f -> Neg $ K i f)

type TypeLabel = [(AgentName, AgentType)]

data Meaning = Mng String (QUForm -> QUForm)
instance Show Meaning where
  show (Mng s _) = s
instance Eq Meaning where
  (Mng s1 _) == (Mng s2 _) = s1 == s2
instance Ord Meaning where
  compare (Mng s1 _) (Mng s2 _) = compare s1 s2
```

```
type UtteranceLabel = [(Utterance,Meaning)]

data Model = Mo
  [(World, (Label, TypeLabel, UtteranceLabel))]
  [(AgentName, Partition World)]
type Scene = (Model, World)

instance Show Model where
  show (Mo ws pss) =
    concatMap showW ws ++ concatMap showP pss
      where
        showW (w,(ps,ts,us)) = concat
          [ show w, ": ", show ps, ", ", showTS ts, ", ", showUS us, "\n" ]
        showP (a, p) = a ++ ": " ++ show p ++ "\n"
        showT (a, AgT agts _) = a ++ ":" ++ agts
        showTS ts = unwords $ map showT ts
        showU (u, mng) = u ++ "=" ++ show mng
        showUS us = unwords $ map showU us

third :: (a,b,c) -> c
third (_,_,z) = z

labelAt :: Scene -> Label
labelAt (Mo ws _, w) = l where Just (l,_,_) = lookup w ws

typeAtOf :: Scene -> AgentName -> AgentType
typeAtOf (Mo ws _, w) a = t where
  Just (_,tOf,_) = lookup w ws
  Just t = lookup a tOf

meaningAtOfGiven :: Scene -> Utterance -> Meaning
meaningAtOfGiven (Mo ws _, w) u = m where
  val = fromMaybe
    (error $ "World " ++ show w ++ " disappeared!" ++ show ws)
    (lookup w ws)
  Just m = lookup u (third val)

meanfctAtOfGiven :: Scene -> Utterance -> QUForm -> QUForm
meanfctAtOfGiven sc u = mngf where
  Mng _ mngf = meaningAtOfGiven sc u

partOf :: Model -> AgentName -> Partition World
partOf (Mo _ pss) i = ps where Just ps = lookup i pss

partAtOf :: Scene -> AgentName -> [World]
partAtOf (Mo _ pss, w) i = head $ filter (elem w) ps where
  Just ps = lookup i pss
```

3.2 Semantics

Definition 7: PQLTTU Semantics (p. 142)

```
type Context = Maybe (AgentName,QUForm)

qIsTrue :: Scene -> Context -> QUForm -> Bool
qIsTrue _  _ Top          = True
qIsTrue sc _ (Prp p)      = p `elem` labelAt sc
qIsTrue sc _ (a `IsA` t)  = typeAtOf sc a == t
qIsTrue sc c (Neg f)      = not (qIsTrue sc c f)
qIsTrue sc c (Conj fs)    = all (qIsTrue sc c) fs
qIsTrue sc c (Disj fs)    = any (qIsTrue sc c) fs
qIsTrue sc c (Impl f g)   = not (qIsTrue sc c f) || qIsTrue sc c g
qIsTrue (m,w) c (K a f)   = all (\v -> qIsTrue (m,v) c f) (partAtOf (m,w) a)
qIsTrue (m,w) c (C as f) = all (\v -> qIsTrue (m,v) c f) vs where
  vs    = head $ filter (elem w) ckrel
  ckrel = fusion $ concat [ partOf m i | i <- as ]
```

Note that the last clause is not in [LW13]. As usual we interpret common knowledge as the transitive closure of the union of the epistemic relations of a set of agents. Because these

relations are represented as partitions we use the `fusion` function from [vE14].

```
qIsTrue sc _ (Ask a f g) = qIsTrue sc (Just (a,f)) g
qIsTrue _         Nothing (Say {}) = error "Nobody asked anything!"
qIsTrue sc@(m,w) c@(Just (a1,q)) (Say a2 u phi) =
  (a1 == a2) && f `elem` [q,Neg q] && qCanSay sc a1 f
    `implies` qIsTrue (qAnnounce m c a1 u, w) Nothing phi
  where
    f = meanfctAtOfGiven sc u q
```

Now the last clause quantifies over all utterances. This is easier to deal with than the quantifier over all formulas above: The model gives us a list of all interpretable utterances.

```
qIsTrue sc@(m,_) c (Ans a phi) =
  all (\u -> qIsTrue sc c (Say a u phi)) relevantUtterances where
    relevantUtterances :: [Utterance]
    relevantUtterances = case c of
      Nothing -> []
      Just _  -> allUtterances m
    allUtterances (Mo ws _) = map fst $ third $ snd $ head ws

qCanSay :: Scene -> AgentName -> QUForm -> Bool
qCanSay sc a f = qIsTrue sc Nothing (agtf a f) where AgT _ agtf = typeAtOf sc a
```

The function to restrict a model now takes an utterance, not a formula as argument. This is important because the utterance can have different meanings in other worlds in which case also the formula giving the condition whether the world is kept or deleted differs from the one at the actual world.

```
qAnnounce :: Model -> Context -> AgentName -> Utterance -> Model
qAnnounce oldm@(Mo oldws oldpss) c a u = Mo ws pss where
  ws = filter check oldws
  check (v,_) = qIsTrue (oldm,v) Nothing (agtf a f) where
    AgT _ agtf = typeAtOf (oldm,v) a
    f = meanfctAtOfGiven (oldm,v) u q
    Just (_,q) = c
  pss = map (fmap strip) oldpss
  strip set = filter ([]/=) $ map (filter (`elem` map fst ws)) set
```

Validity is truth at all worlds:

```
qIsValid :: Model -> QUForm -> Bool
qIsValid m@(Mo ws _) f = and [ qIsTrue (m,w) Nothing f | w <- map fst ws ]
```

3.3 Questioning Strategies, Puzzles and Solutions

Definition 8: Questioning Strategy (p. 143)
We define a question as a pair of an agent and a formula. A questioning strategy is a directed graph where the nodes are questions and the edges are labeled with utterances.
As in the paper we only use strategies which are directed acyclic graphs (DAGs) here. Thus a strategy is either a final state (stop asking) or a question state which comes with the information whom to ask what and a map of utterances to follow-up strategies (what to do next, depending on the utterance).

```
type Question = (AgentName,QUForm)
type Answer = Utterance
data Strategy = FState | QState Question [(Utterance,Strategy)] deriving (Eq)

instance Show Strategy where
  show = myshow "" where
    myshow space FState = space ++ "Stop.\n"
    myshow space (QState (a,f) branches) =
      space ++ "Ask " ++ a ++ " if >>" ++ show f ++ "<<.\n"
        ++ concatMap linesFor branches
      where
        linesFor (u,FState) = space ++ "  If they say '" ++ show u ++ "', then stop.\n"
```

```
        linesFor (u,nexts ) = space ++ "  If they say '" ++ show u ++ "', then:\n"
          ++ myshow (space ++ "    ") nexts
```

```
type Sequence = [(Question,Answer)]

sequencesOf :: Strategy -> [Sequence]
sequencesOf FState = [ [] ]
sequencesOf (QState q branches) =
  concatMap (\(u,next) -> [(q,u):rest | rest <- sequencesOf next]) branches
```

A puzzle is a pair (M, θ) of a model which describes the initial situation and a formula which states the goal to be reached. It is solved by a strategy S iff for all sequences s of S we have that $M \vDash \varphi_s^\theta$. Here φ_s^θ is the formula saying that all reachable questions in s can be answered and after the whole sequence is executed θ holds.

```
type Puzzle = (Model,QUForm)

solves :: Strategy -> Puzzle -> Bool
solves strategy (m,goal) = all
  (qIsValid m . seq2f goal)
  (sequencesOf strategy)

seq2f :: QUForm -> Sequence -> QUForm
seq2f goal (((a,q),u):rest) =
  Ask a q $                        -- After asking a whether q,
    Conj [ diamAns a Top           --   a can answer somehow and
         , Say a u $ seq2f goal rest ] --   after a says u ...
seq2f goal [] = goal               -- ... the goal formula is true.
```

If the questions in a strategy are always answerable, the check whether a strategy solves a puzzle can be simplified by leaving out `diamAns a Top`.

```
solvesSimple :: Strategy -> Puzzle -> Bool -- p. 144
solvesSimple strategy (model,goal) = all okay (sequencesOf strategy) where
  okay qaSeq = qIsValid model (seq2fSimple qaSeq)
  seq2fSimple [] = goal
  seq2fSimple (((a,q),u):rest) = Ask a q $ Say a u $ seq2fSimple rest
```

3.4 Formalizing HLPE

We are now on page 145.

```
allTypes :: [AgentType]
allTypes = [truthTeller,liar,bluffer]

allAgents :: [AgentName]
allAgents = ["A","B","C","D"]

knights :: [AgentName]
knights = ["A","B","C"]

jaInt,daInt :: UtteranceLabel
jaInt = [("ja", Mng "yes" id), ("da", Mng "no" Neg)]
daInt = [("ja", Mng "no" Neg), ("da", Mng "yes" id)]

m0 :: Model
m0 = Mo
    (zip [1..] allVals)
    [ ("A", knowAll), ("B", knowAll), ("C", knowAll), ("D", knowNothing)]
  where
    typeLists = permutations [truthTeller,liar,bluffer]
    agLabelFor [at,bt,ct] = [("A",at), ("B",bt), ("C",ct), ("D",truthTeller)]
    -- the paper does not give a type to "D" but it is needed to evaluate phiE5
    agLabelFor _ = error "this will never happen."
    allVals = [ ([],agLabelFor ats,uLabel) | uLabel <- [jaInt,daInt], ats <- typeLists ]
    knowAll = map return [1..(length allVals)]
    knowNothing = [ [1..(length allVals)] ]
```

```
*QULogic GOA> m0
1:  [], A:TT B:LL C:LT D:TT, ja=yes da=no
2:  [], A:LL B:TT C:LT D:TT, ja=yes da=no
3:  [], A:LT B:LL C:TT D:TT, ja=yes da=no
4:  [], A:LL B:LT C:TT D:TT, ja=yes da=no
5:  [], A:LT B:TT C:LL D:TT, ja=yes da=no
6:  [], A:TT B:LT C:LL D:TT, ja=yes da=no
7:  [], A:TT B:LL C:LT D:TT, ja=no  da=yes
8:  [], A:LL B:TT C:LT D:TT, ja=no  da=yes
9:  [], A:LT B:LL C:TT D:TT, ja=no  da=yes
10: [], A:LL B:LT C:TT D:TT, ja=no  da=yes
11: [], A:LT B:TT C:LL D:TT, ja=no  da=yes
12: [], A:TT B:LT C:LL D:TT, ja=no  da=yes
A: [[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]]
B: [[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]]
C: [[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]]
D: [[1,2,3,4,5,6,7,8,9,10,11,12]]
```

3.5 Formalizing and checking the modelling assumptions

```
phiE1, phiE2, phiE3, phiE4, phiE5 :: QUForm
-- A, B and C are of different types and this is common knowledge.
phiE1 = C allAgents knightsDiff where
  knightsDiff = Conj [ differentTs a1 a2 | a1 <- knights, a2 <- delete a1 knights ]
  differentTs a1 a2 = Conj [ (a1 `IsA` t) `Impl` (Neg $ a2 `IsA` t) | t <- allTypes ]
-- A, B and C know each other's type and this is common knowledge.
phiE2 = C allAgents knightsKnow where
  knightsKnow = Conj [ a1 `knowsTypeOf` a2 | a1 <- knights, a2 <- knights ]
  knowsTypeOf a1 a2 = Disj [ K a1 (a2 `IsA` t) | t <- allTypes ]
-- A, B and C know the meaning of 'da' and 'ja' and this is common knowledge.
phiE3 = C allAgents knightsKnow where
  knightsKnow = Conj [ Disj [ K k jaMeansYes, K k daMeansYes ] | k <- knights ]
-- D does not know the types of A, B, C and this is common knowledge.
phiE4 = C allAgents $ Conj
  [ Neg $ K "D" (k `IsA` t) | k <- ["A","B","C"], t <- allTypes ]
-- D does not know the meanings of 'da; and 'ja' but he knows that one
-- means 'yes' and the other means 'no', and this is common knowledge.
phiE5 = C allAgents $ Conj
  [ K "D" $ Disj [ jaMeansYes, daMeansYes ]
  , Neg $ Disj [ K "D" jaMeansYes, K "D" daMeansYes ] ]

jaMeansYes :: QUForm
jaMeansYes = Conj (map phiJA allAgents) where
  phiJA x = (x `IsA` truthTeller) `Impl` Conj
    [ Ask x (      x `IsA` truthTeller) $ diamSay x "ja" Top
    , Ask x (Neg $ x `IsA` truthTeller) $ diamSay x "da" Top ]

daMeansYes :: QUForm
daMeansYes = Conj (map phiDA allAgents) where
  phiDA x = (x `IsA` truthTeller) `Impl` Conj
    [ Ask x (      x `IsA` truthTeller) $ diamSay x "da" Top
    , Ask x (Neg $ x `IsA` truthTeller) $ diamSay x "ja" Top ]
```

Check that the model m0 satisfies all the modelling assumptions:

```
*QULogic GOA> qIsValid m0 (Conj [phiE1,phiE2,phiE3,phiE4,phiE5])
True
```

We can now define the HLPE itself:

```
hlpe :: Puzzle
hlpe = (m0, chi) where
  chi = Conj [ Disj [ K "D" (a `IsA` t) | t <- allTypes ] | a <- knights ]
```

The E^* operator:

```
eStar :: AgentName -> QUForm -> QUForm
eStar a f = Ask a f $ diamSay a "ja" Top
```

3.6 Checking the Rabern Strategy

On page page 147 of [LW13] we find the following formula, describing how D can learn something no matter how B answers the nested question.

```
checkHint :: QUForm
checkHint = Ask "B"
  (Ask "B" ("A" `IsA` bluffer) $ diamSay "B" "ja" Top)
  (Conj
    [ Say "B" "ja" $ K "D" ( (Neg $ "B" `IsA` bluffer) `Impl` ("A" `IsA` bluffer) )
    , Say "B" "da" $ K "D" ( (Neg $ "B" `IsA` bluffer) `Impl` (Neg $ "A" `IsA` bluffer) )
        ])
```

```
*QULogic GOA> qIsValid m0 checkHint
True
```

The strategy from Rabern and Rabern [RR08], as formalized on page 148 of [LW13].

```
rabern :: Strategy
rabern =
  QState q1
    [ ("ja", QState q2c [ (u, QState q3c stop ) | u <- [ "ja", "da" ] ] )
    , ("da", QState q2a [ (u, QState q3a stop ) | u <- [ "ja", "da" ] ] ) ]
  where
    -- First ask B to find out whether A or C is a non-bluffer:
    q1  = ("B", eStar "B" ("A" `IsA` bluffer))
    -- Now ask the non-bluffer about the other two:
    q2c = ("C", eStar "C" ("C" `IsA` truthTeller))
    q3c = ("C", eStar "C" ("B" `IsA` bluffer))
    q2a = ("A", eStar "A" ("A" `IsA` truthTeller))
    q3a = ("A", eStar "A" ("B" `IsA` bluffer))
    stop = [ (u, FState) | u <- [ "ja", "da" ] ]
```

```
*QULogic> rabern `solves` hlpe
True
```

3.7 Replacing gods with ordinary people

We now implement Example 6: HLPE with ignorance.

```
allSubjTypes :: [AgentType]
allSubjTypes = [subjTruthTeller,subjLiar,bluffer]

phiE0',phiE1',phiE2',phiE3',phiE4',phiE5' :: QUForm
-- A, B and C are of types STT,SLL,LT and this is common knowledge (atick).
phiE0' = C allAgents $ Conj (map someoneIs allSubjTypes) where
  someoneIs t = Disj [ a `IsA` t | a <- knights ]
-- A, B and C are of different types atick.
-- (This is the same as phiE1 above, but redefined here for the subjective types.
--  Could be avoided if we had a primitive operator for "have different types")
phiE1' = C allAgents knightsDiff where
  knightsDiff = Conj [ differentTs a1 a2 | a1 <- knights, a2 <- delete a1 knights ]
  differentTs a1 a2 = Conj [ (a1 `IsA` t) `Impl` (Neg $ a2 `IsA` t) | t <- allSubjTypes ]
-- A, B and C know their own types but do not know others' types, atick.
phiE2' = C allAgents $ Conj $ map knowSelf knights ++ map notknowOthers knights where
  knowSelf a = Disj [ K a (a `IsA` t) | t <- allSubjTypes ]
  notknowOthers a = Conj [Neg $ Disj [ K a (b `IsA` t) | t <- allSubjTypes ] | b <- delete
      a knights]
-- A, B and C know the meaning of 'da' and 'ja' and this is common knowledge.
phiE3' = phiE3
-- D does not know the types of A, B, C and this is common knowledge.
phiE4' = C allAgents $ Conj
  [ Neg $ K "D" (k `IsA` t) | k <- ["A","B","C"], t <- allSubjTypes ]
-- D does not know the meanings of 'da; and 'ja' but he knows that one
-- means 'yes' and the other means 'no', and this is common knowledge.
phiE5' = C allAgents $ Conj
  [ K "D" $ Disj [ jaMeansYes, daMeansYes ]
  , Neg $ Disj [ K "D" jaMeansYes, K "D" daMeansYes ] ]
```

Then we define the new model M_1 as follows.

```
m1 :: Model
m1 = Mo states (map (\c -> ([c],relFor [c])) "ABCD")
  where
    typeLists = permutations allSubjTypes
    agLabelFor [at,bt,ct] = [("A",at), ("B",bt), ("C",ct), ("D",truthTeller)]
    agLabelFor _ = error "this will never happen."
    allVals = [ ([],agLabelFor ats,uLabel) | uLabel <- [jaInt,daInt], ats <- typeLists ]
    states = zip [1..] allVals
    relFor "D" = [ [1..(length allVals)] ] -- knows nothing
    relFor a = sort $ map (map fst) $ groupBy same (sortOn sortpick states) where
      same (_,(_,sTl,sUl)) (_,(_,tTl,tUl)) = sTl!a == tTl!a && sUl == tUl
      sortpick (_,(_,sTl,sUl)) =  (sUl,sTl!a)
```

```
*QULogic GOA> m1
1:  [], A:TT B:LL C:LT D:TT, ja=yes da=no
2:  [], A:LL B:TT C:LT D:TT, ja=yes da=no
3:  [], A:LT B:LL C:TT D:TT, ja=yes da=no
4:  [], A:LL B:LT C:TT D:TT, ja=yes da=no
5:  [], A:LT B:TT C:LL D:TT, ja=yes da=no
6:  [], A:TT B:LT C:LL D:TT, ja=yes da=no
7:  [], A:TT B:LL C:LT D:TT, ja=no  da=yes
8:  [], A:LL B:TT C:LT D:TT, ja=no  da=yes
9:  [], A:LT B:LL C:TT D:TT, ja=no  da=yes
10: [], A:LL B:LT C:TT D:TT, ja=no  da=yes
11: [], A:LT B:TT C:LL D:TT, ja=no  da=yes
12: [], A:TT B:LT C:LL D:TT, ja=no  da=yes
A: [[1,6],[2,4],[3,5],[7,12],[8,10],[9,11]]
B: [[1,3],[2,5],[4,6],[7,9],[8,11],[10,12]]
C: [[1,2],[3,4],[5,6],[7,8],[9,10],[11,12]]
D: [[1,2,3,4,5,6,7,8,9,10,11,12]]
```

And check that it satisfies all the modelling assumptions:

```
*QULogic GOA> all (qIsValid m1) [phiE0', phiE1', phiE2', phiE3', phiE4', phiE5']
True
```

We can now check the formula from page 150 to see that Lemma 1 does not hold on `m1`.

```
lemma1broken :: QUForm
lemma1broken = Ask "A"
  (Ask "A" ("B" `IsA` bluffer) $ diamSay "A" "ja" Top)
  (Say "A" "da" $
    K "D" ( (Neg $ "A" `IsA` bluffer) `Impl` (Neg $ "B" `IsA` bluffer) ) )
```

```
*Explain GOA> qIsValid m0 lemma1broken
True
*Explain GOA> qIsValid m1 lemma1broken
False
```

We now define the unsolvable puzzles from page 151. Note that our program provides no way of checking that there is no solution. One way to realize this would be to introduce an "arbitrary question" operator, saying "there is a question after which ...".

```
subjUnsolvableHlpe :: Puzzle
subjUnsolvableHlpe = (m2,goal) where
  goal = Conj (map theta knights) where
    theta a = Disj [ K "D" (a `IsA` t) | t <- allSubjTypes ]
```

However, the following more modest goal is reachable.

```
subjGoal' :: QUForm
subjGoal' = Disj (map theta' knights) where
  theta' a = Disj [ K "D" (a `IsA` t) | t <- [subjLiar,subjTruthTeller] ]
```

Defining m2 as the upper part of m1:

```
m2 :: Model
m2 = Mo newstates newrel where
  Mo oldstates oldrel = m1
  newstates = filter (\(_,(_,_,ulabel)) -> jaInt==ulabel) oldstates
```

```
  newrel = map (second restrict) oldrel
  restrict r = filter (not.null) $ map (filter (`elem` map fst newstates)) r
```

The model looks as follows and it is easy to verify that the meaning of "ja" is common knowledge here.

```
*QULogic GOA> m2
1: [], A:STT B:SLL C:LT  D:TT, ja=yes da=no
2: [], A:SLL B:STT C:LT  D:TT, ja=yes da=no
3: [], A:LT  B:SLL C:STT D:TT, ja=yes da=no
4: [], A:SLL B:LT  C:STT D:TT, ja=yes da=no
5: [], A:LT  B:STT C:SLL D:TT, ja=yes da=no
6: [], A:STT B:LT  C:SLL D:TT, ja=yes da=no
A: [[1,6],[2,4],[3,5]]
B: [[1,3],[2,5],[4,6]]
C: [[1,2],[3,4],[5,6]]
D: [[1,2,3,4,5,6]]

*QULogic GOA> qIsValid m2 (C allAgents jaMeansYes)
True
```

The solution from page 152 is to as ask each of A, B and C "Are you a bluffer". In this strategy the answers do not matter to which question is asked next. The following code highlights this with the infix function `` `anyway` ``. Moreover, we can also check the strategy with nested question to reach the goal on m1.

```
m2strat :: Strategy
m2strat = ask "A" `anyway` ( ask "B" `anyway` ( ask "C" `anyway` FState ) ) where
  anyway q s = QState q [ (u,s) | u <- [ "ja", "da" ] ]
  ask i  = (i, i `IsA` bluffer)

m1strat :: Strategy
m1strat = ask "A" `anyway` ( ask "B" `anyway` ( ask "C" `anyway` FState ) ) where
  anyway q s = QState q [ (u,s) | u <- [ "ja", "da" ] ]
  ask i  = (i, nest i (i `IsA` bluffer))
  nest i psi = Ask i
    (Conj [ i `IsA` subjTruthTeller `Impl` psi, i `IsA` subjLiar `Impl` Neg psi])
    (diamSay i "ja" Top)
```

```
*QULogic GOA> m2strat `solves` (m2,subjGoal')
True
*QULogic GOA> m1strat `solves` (m1,subjGoal')
True
```

References

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.

[RR08] Brian Rabern and Landon Rabern. A simple solution to the hardest logic puzzle ever. *Analysis*, 68(2):105–112, 2008.

[vE07] Jan van Eijck. DEMO—a demo of epistemic modelling. In *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London*, volume 1, pages 303–362, 2007.

[vE14] Jan van Eijck. Relations, Equivalences, Partitions. Technical report, CWI, 2014.