Demo Light for Composing Models

Jan van Eijck

with contributions by Lakshmanan Kuppusamy and Floor Sietsma

October 11, 2011

Abstract

Light version of DEMO for composing epistemic models, based on the code for the ESSLLI 2008 course on Dynamic Epistemic Logic (see `http://homepages.cwi.nl/~jve/courses/esslli08/`) extended with vocabulary information [EWS10]. Factual change is also treated and the models are extended with integer registers. There are some examples: the muddy children, and hat puzzles, dealing with the interaction of perception and change [Eijar]. The piece ends with an analysis of the game of Liar's Dice in the spirit of [DvESW07].

Contents

```
module DemoLight

where
import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
import ChangeVocab
import ChangePerception
```

Chapter 1

Models with Vocabulary

Module declaration. We will use QuickCheck [CH00] for some simple tests.

```
module ModelsVocab where

import List
import Test.QuickCheck
```

Binary relations as lists of ordered pairs:

```
type Rel a = [(a,a)]
```

Test for equality of relations:

```
sameR :: Ord a => Rel a -> Rel a -> Bool
sameR r s = sort (nub r) == sort (nub s)
```

Operations on relations: converse Relational converse $R^\smile$ is given by:

$$R^\smile = \{(y, x) \mid (x, y) \in R\}$$

```
cnv :: Rel a -> Rel a
cnv r = [ (y,x) | (x,y) <- r ]
```

Operations on relations: composition The relational composition of two relations R and S on a set A:

$$R \circ S = \{(x, z) \mid \exists y \in A(xRy \wedge ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Note that `(@@)` is the prefix version of `@@`.

Testing for Euclideanness

A relation R is euclidian if $\forall xyz((Rxy \wedge Rxz) \rightarrow Ryz)$.

In other words: R is euclidean iff $R^\smile \circ R \subseteq R$.

Use this for a test of Euclideanness:

$$A \subseteq B \ :\equiv \ \forall x \in A : x \in B.$$

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = all (\ x -> elem x ys) xs
```

```
euclR :: Eq a => Rel a -> Bool
euclR r = (cnv r @@ r) `containedIn` r
```

Test for Seriality

A relation R is *serial* if $\forall x \exists y Rxy$ holds.

Here is a test:

```
serialR :: Eq a => Rel a -> Bool
serialR r =
  all (not.null)
    (map (\ (x,y) -> [ v | (u,v) <- r, y == u]) r)
```

Testing for S5 An accessibility relation is S5 if it is an equivalence.

```
reflR :: Eq a => [a] -> Rel a -> Bool
reflR xs r =
  [(x,x) | x <- xs] `containedIn` r

symmR :: Eq a => Rel a -> Bool
symmR r = cnv r `containedIn` r

transR :: Eq a => Rel a -> Bool
transR r = (r @@ r) `containedIn` r

isS5 :: Eq a => [a] -> Rel a -> Bool
isS5 xs r = reflR xs r && transR r && symmR r
```

Testing for KD45

An accessibility relation is KD45 if it is serial, transitive and euclidean.

```
isKD45 :: Eq a => Rel a -> Bool
isKD45 r = transR r && serialR r && euclR r
```

Representing Epistemic Models: Agents

An infinite number of agents, with names for the first five of them:

```
data Agent = Ag Int deriving (Eq,Ord)

a,alice, b,bob, c,carol, d,dave, e,ernie  :: Agent
a = Ag 0; alice = Ag 0
b = Ag 1; bob   = Ag 1
c = Ag 2; carol = Ag 2
d = Ag 3; dave  = Ag 3
e = Ag 4; ernie = Ag 4

instance Show Agent where
  show (Ag 0) = "a"; show (Ag 1) = "b";
  show (Ag 2) = "c"; show (Ag 3) = "d" ;
  show (Ag 4) = "e";
  show (Ag n) = 'a': show n
```

Representing Epistemic Models: Basic Propositions

```
data Prp = P Int | Q Int | R Int deriving (Eq,Ord)

instance Show Prp where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
```

Representing Epistemic Models: Basic Propositions

Registers are variables that can hold a number.

```
data Reg = Rg Int deriving (Eq,Ord)

instance Show Reg where
  show (Rg n)       = 'R': show n
```

A Datatype for Epistemic Models

The model has a vocabulary of propositions and registers. Each state has a list of propositions that are true and a list of registers and their values.

```
data EpistM state = Mo
             [state]
             [Agent]
             [Prp]
             [Reg]
             [(state,[Prp])]
             [(state,[(Reg, Int)])]
             [(Agent,state,state)]
             [state]  deriving (Eq,Show)
```

Example Epistemic Model

```
s5example :: EpistM Integer
s5example =
 Mo [0..3]
    [a,b,c]
    [P 0, Q 0]
    [Rg 0]
    [(0,[]),(1,[P 0]),(2,[Q 0]),(3,[P 0, Q 0])]
    [(0,[(Rg 0, 0)]), (1,[(Rg 0, 1)]),
     (2,[(Rg 0, 2)]), (3,[(Rg 0, 3)])]
    ([ (a,x,x) | x <- [0..3] ] ++
     [ (b,x,x) | x <- [0..3] ] ++
     [ (c,x,y) | x <- [0..3], y <- [0..3] ])
    [1]
```

Extracting domain, vocabulary, relations, and valuation from an epistemic model

```
dom :: EpistM a -> [a]
dom (Mo states _ _ _ _ _ _ _) = states

rel :: Agent -> EpistM a -> Rel a
rel ag (Mo _ _ _ _ _ _ rels _) =
      [ (x,y) | (agent,x,y) <- rels, ag == agent ]

valuationProp ::  EpistM a -> [(a,[Prp])]
valuationProp (Mo _ _ _ _ valprop _ _ _ ) = valprop

valuationReg ::  EpistM a -> [(a,[(Reg, Int)])]
valuationReg (Mo _ _ _ _ _ valreg _ _ ) = valreg

vocabProp ::  EpistM a -> [Prp]
vocabProp  (Mo _ _ props _ _ _ _ _) = props

vocabReg ::  EpistM a -> [Reg]
vocabReg  (Mo _ _ _ regs _ _ _ _) = regs

valPropStat :: Eq a => a -> EpistM a -> [Prp]
valPropStat x (Mo _ _ _ _ valprop _ _ _)
   = [z | (state,y) <- valprop, state == x, z <- y]

valRegStat :: Eq a => a -> EpistM a -> [(Reg, Int)]
valRegStat x (Mo _ _ _ _ _ valreg _ _)
   = [z | (state,y) <- valreg, state == x, z <- y]

valRegSet :: Eq a => EpistM a -> [(Reg, Int)]
valRegSet  (Mo _ _ _ _ _ valreg _ _)
       = nub ( [y | (state,x) <- valreg, y <- x] )

vcbPropSet :: Eq a => EpistM a -> [Prp]
vcbPropSet  (Mo _ _ _ _ valprop _ _ _)
       = nub ( [y | (state,x) <- valprop, y <- x] )

vcbRegSet :: Eq a => EpistM a -> [Reg]
vcbRegSet  (Mo _ _ _ _ _ valreg _ _)
       = nub ( [y | (state,x) <- valreg, (y,z) <- x] )

actual :: EpistM a -> [a]
actual (Mo _ _ _ _ _ _ _ actual) = actual
```

From equivalence relations to partitions Every equivalence relation R on A corresponds to a partition on A: the set $\{[a]_R \mid a \in A\}$, where $[a]_R = \{b \in A \mid (a, b) \in R\}$.

```
rel2partition :: Ord a => [a] -> Rel a -> [[a]]
rel2partition [] r = []
rel2partition (x:xs) r =
   xclass :  rel2partition (xs \\ xclass) r
     where
       xclass = x : [ y | y <- xs, elem (x,y) r ]
```

Displaying S5 Models The function `rel2partition` can be used to write a display function for S5 models that shows each accessibility relation as a partition, as follows.

```
showS5 :: (Ord a,Show a) => EpistM a -> [String]
showS5 m@(Mo states agents props regs
             valprop valreg rels actual) =
  show states    :
  show props :
  show regs  :
  show valprop  :
  show valreg   :
  map show [ (ag, (rel2partition states) (rel ag m))
                                      | ag <- agents ]
  ++
  [show actual]
```

Here `@` is used to introduce a shorthand or name for a datastructure.

```
displayS5 ::  (Ord a,Show a) => EpistM a -> IO()
displayS5 = putStrLn . unlines . showS5
```

Blissful Ignorance Blissful ignorance is the state where you don't know anything, but you know also that there is no reason to worry, for you know that nobody knows anything.

A Kripke model where every agent from agent set A is in blissful ignorance about a (finite) set of propositions P, with $|P| = k$, with no registers, looks as follows:

$$M = (W, V, R) \text{ where}$$

$$\begin{array}{lcl} W & = & \{0, \ldots, 2^k - 1\} \\ V & = & \text{any surjection in } W \to \mathcal{P}(P) \\ R & = & \{x \xrightarrow{a} y \mid x, y \in W, a \in A\}. \end{array}$$

Note that V is in fact a bijection, for $|\mathcal{P}(P)| = 2^k = |W|$.

Generating Models for Blissful Ignorance

```
initM :: [Agent] -> [Prp] -> EpistM Integer
initM ags props = (Mo worlds ags props regs
                      valprop valreg accs points)
  where
    worlds = [0..(2^k-1)]
    regs = []
    k      = length props
    valprop    = zip worlds (sortL (powerList props))
    valreg = [(w,[]) | w <- worlds]
    accs   = [ (ag,st1,st2) | ag  <- ags,
                              st1 <- worlds,
                              st2 <- worlds      ]
    points = worlds
```

The model ϵ for blissful ignorance with an empty vocabulary:

```
epsilon :: [Agent] -> EpistM Integer
epsilon ags = initM ags []
```

powerList, sortL (sort by length)

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

sortL :: Ord a => [[a]] -> [[a]]
sortL  = sortBy
  (\ xs ys -> if length xs < length ys
                then LT
              else if length xs > length ys
                then GT
              else compare xs ys)
```

General Knowledge The general knowledge accessibility relation of a set of agents C is given by

$$\bigcup_{c \in C} R_c.$$

```
genK :: Ord state => [(Agent,state,state)]
                      -> [Agent] -> Rel state
genK r ags = [ (x,y) | (ag,x,y) <- r, ag `elem` ags ]
```

Right Section of a Relation

If R is a binary relation on A, and $a \in A$, then aR is the set

$$\{b \in A \mid aRb\}.$$

```
rightS :: Ord a => Rel a -> a -> [a]
rightS r x = (sort.nub) [ z | (y,z) <- r, x == y ]
```

General Knowledge Alternatives

```
genAlts :: Ord state => [(Agent,state,state)]
        -> [Agent] -> state -> [state]
genAlts r ags = rightS (genK r ags)
```

Closures of Relations If $\mathcal{O}$ is a set of properties of relations on a set A, then the $\mathcal{O}$ closure of a relation R on A is *the smallest relation S that includes R and that has all the properties in $\mathcal{O}$.*

The closures of relations that we need are the transitive closure and the reflexive transitive closure.

Reflexive Transitive Closure Let a set A be given. Let R be a binary relation on A. Let $I = \{(x, x) \mid x \in A\}$.

We define R^n for $n \geq 0$, as follows:

- $R^0 = I$.
- $R^{n+1} = R \circ R^n$.

Next, define R^* by means of:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n.$$

Computing Reflexive Transitive Closure If A is finite, any R on A is finite as well. In particular, there will be k with $R^{k+1} \subseteq R^0 \cup \cdots \cup R^k$.

Thus, in the finite case reflexive transitive closure can be computed by successively computing $\bigcup_{n \in \{0,..,k\}} R^n$ until $R^{k+1} \subseteq \bigcup_{n \in \{0,..,k\}} R^n$.

In other words: the reflexive transitive closure of a relation R can be computed from I by repeated application of the operation

$$\lambda S.(S \cup (R \circ S)),$$

until the operation reaches a fixpoint. A more efficient computation of the reflexive transitive closure of R is by repeated application of the operation

$$\lambda S.(S \cup (S \circ S)),$$

starting from $I \cup R$, until the operation reaches a fixpoint.

Least Fixpoint A fixpoint of an operation f is an x for which $f(x) = x$. Least fixpoint calculation:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Computing Reflexive Transitive Closure, at last:

```
rtc :: Ord a => [a] -> Rel a -> Rel a
rtc xs r = lfp (\ s -> (sort.nub) (s ++ (s@@s))) ri
  where ri = nub (r ++ [(x,x) | x <- xs ])
```

Some test properties:

```
prop_RtcRefl ::  Ord a => [a] -> Rel a -> Bool
prop_RtcRefl xs r = reflR xs (rtc xs r)
```

```
prop_RtcTr ::  Ord a => [a] -> Rel a -> Bool
prop_RtcTr xs r = transR (rtc xs r)
```

Computing Transitive Closure Same as computing reflexive transitive closure, but starting out from the relation R.

```
tc :: Ord a => Rel a -> Rel a
tc r = lfp (\ s -> (sort.nub) (s ++ (s @@ s))) r
```

A test property:

```
prop_TcTr ::  Ord a => Rel a -> Bool
prop_TcTr r = transR (tc r)
```

Computing Common Knowledge The common knowledge relation for group of agents C is the relation

$$(\bigcup_{c \in C} R_c)^*.$$

Given that the R_c are represented as a list of triples

`[(Agent,state,state)]`

we can define a function that extracts the common knowledge relation:

```
commonK :: Ord state => [(Agent,state,state)]
           -> [Agent] -> [state] -> Rel state
commonK r ags xs = rtc xs (genK r ags)
```

Common Knowledge Alternatives

```
commonAlts :: Ord state => [(Agent,state,state)]
        -> [Agent] -> [state] -> state -> [state]
commonAlts r ags xs s = rightS (commonK r ags xs) s
```

Representing Formulas

A formula can be about propositions or about arithmetic expressions, which can be compared.

```
data Form = Top
          | Prp Prp
          | Gt Arith Arith
  | Eq Arith Arith
          | Neg Form
          | Conj [Form]
          | Disj [Form]
          | K Agent Form
          | CK [Agent] Form
          deriving (Eq,Ord)
```

CK is the operator for common knowledge.

Arithmetic expressions are integers, registers or the sum of arithmetic expressions:

```
data Arith = I Int
           | Reg Reg
           | ASum [Arith]
           deriving (Eq,Ord)
```

Example formulas

```
p = Prp (P 0)
q = Prp (Q 0)
```

```
instance Show Form where
  show Top           = "T"
  show (Gt a b)     = '>': show a ++ show b
  show (Eq a b)     = '=': show a ++ show b
  show (Prp p)      = show p
  show (Neg f)       = '-':(show f)
  show (Conj fs)     = '&': show fs
  show (Disj fs)     = 'v': show fs
  show (K agent f)   = '[':show agent++"]"++show f
  show (CK agents f) = 'C': show agents ++ show f
```

```
instance Show Arith where
  show (I i)        = show i
  show (Reg r)      = show r
  show (ASum as)    = '+': show as
```

Getting the proposition letters from a formula:

```
getPs :: Form -> [Prp]
getPs Top     = []
getPs (Gt a b) = []
getPs (Eq a b) = []
getPs (Prp p) = [p]
getPs (Neg f) = getPs f
getPs (Conj fs) = (sort.nub.concat) (map getPs fs)
getPs (Disj fs) = (sort.nub.concat) (map getPs fs)
getPs (K agent f)   = getPs f
getPs (CK agents f) = getPs f
```

Getting the registers from a formula:

```
getRs :: Form -> [Reg]
getRs Top     = []
getRs (Gt a b) = getRsA a ++ getRsA b
getRs (Eq a b) = getRsA a ++ getRsA b
getRs (Prp p) = []
getRs (Neg f) = getRs f
getRs (Conj fs) = (sort.nub.concat) (map getRs fs)
getRs (Disj fs) = (sort.nub.concat) (map getRs fs)
getRs (K agent f)   = getRs f
getRs (CK agents f) = getRs f
```

Getting the registers from an arithmetic expression:

```
getRsA :: Arith -> [Reg]
getRsA (I i) = []
getRsA (Reg r) = [r]
getRsA (ASum as) = (sort.nub.concat) (map getRsA as)
```

Valuation Lookup

```
apply :: Eq a => [(a,b)] -> a -> b
apply [] _ = error "argument not in list"
apply ((x,z):xs) y | x == y    = z
                   | otherwise = apply xs y
```

This can be used to look up the valuation for a world in a model.

Maybe types for Booleans and Quantifiers

The following operators implement Strong Kleene evaluation in partial models ([Kle52], Chapter 12, Section 64). The three truth values are: `Nothing`

for 'undefined', `Just True` for 'true', and `Just False` for 'false'. The type for these three values is `Maybe Bool`.

```
maybe_Not :: Maybe Bool -> Maybe Bool
maybe_Not Nothing = Nothing
maybe_Not (Just x) = Just (not x)

maybe_And :: [Maybe Bool] -> Maybe Bool
maybe_And [] = Just True
maybe_And (Nothing:xs) = Nothing
maybe_And ((Just True):xs) = maybe_And (xs)
maybe_And ((Just False):xs) = Just False

maybe_Or :: [Maybe Bool] -> Maybe Bool
maybe_Or [] = Just False
maybe_Or (Nothing:xs) = Nothing
maybe_Or ((Just True):xs) = Just True
maybe_Or ((Just False):xs) = maybe_Or (xs)
```

`maybe_And` can be used for the definition of `maybe_All`, and `maybe_Or` can be used for the definition of `maybe_Any`.

```
maybe_All:: (Maybe Bool -> Maybe Bool) -> [Maybe Bool] ->
            Maybe Bool
maybe_All f = (maybe_And). map f

maybe_Any :: (Maybe Bool -> Maybe Bool) -> [Maybe Bool] ->
             Maybe Bool
maybe_Any f = (maybe_Or). map f
```

Evaluation

We will use this to define partial evaluation, to take into account that a formula may use proposition letters that are not in the vocabulary of a model.

```
isTrueAtMayb :: Ord state =>
     EpistM state -> state -> Form -> Maybe Bool
isTrueAtMayb m w Top = Just True
isTrueAtMayb
 m@(Mo _ _ props _ valprop _ _ _) w (Prp p) =
 if notElem p props then Nothing else
 if elem p (concat [props|(w',props) <- valprop, w'==w]) then
    (Just True)
 else Just False
isTrueAtMayb
 m@(Mo _ _ _ regs _ _ _ _) w (Gt a b) =
 if or (map (\ x -> notElem x regs) ((getRsA a) ++ (getRsA b)))
 then Nothing
 else if arithVal m w a > arithVal m w b then Just True
 else Just False
isTrueAtMayb
 m@(Mo _ _ _ regs _ _ _ _) w (Eq a b) =
 if or (map (\ x -> notElem x regs) ((getRsA a) ++ (getRsA b)))
 then Nothing
 else if arithVal m w a == arithVal m w b then Just True
 else Just False
isTrueAtMayb m w (Neg f)   = maybe_Not (isTrueAtMayb m w f)
isTrueAtMayb m w (Conj fs) =
        maybe_And (map (isTrueAtMayb m w) fs)
isTrueAtMayb m w (Disj fs) =
        maybe_Or  (map (isTrueAtMayb m w) fs)
```

```
isTrueAtMayb
 m w (K ag f) =
 maybe_And (map (flip (isTrueAtMayb m) f)
          (rightS (rel ag m) w))
isTrueAtMayb
 m@(Mo worlds _ _ _ _ _ acc _) w (CK ags f) =
 maybe_And (map (flip (isTrueAtMayb m) f)
          (commonAlts acc ags worlds w))
```

For finding the truth value of formulas of the form $a > b$ we have to know the value of arithmetic values in the model:

```
arithVal :: Ord state => EpistM state -> state -> Arith -> Int
arithVal m w (I i) = i
arithVal (Mo _ _ _ _ _ valreg _ _) w (Reg r) =
  apply (apply valreg w) r
arithVal m w (ASum as) =
  sum (map (arithVal m w) as)
```

`isTrue`, for truth in a model, is also three-valued:

```
isTrue :: Ord state => EpistM state -> Form -> Maybe Bool
isTrue m@(Mo _ _ _ _ _ _ _ points) f =
  maybe_And [isTrueAtMayb m s f | s <- points ]
```

Finally: Public Announcement Update

```
upd_pa :: Ord state =>
          EpistM state -> Form -> EpistM state
upd_pa m@(Mo states agents props regs
          valprop valreg rels actual) f =
  (Mo states' agents props regs
   valprop' valreg' rels' actual')
   where
   states' = [ s | s <- states, isTrueAtMaybb m s f ==
             Just True]
   valprop'    = [(s,p) | (s,p) <- valprop,
                          s `elem` states'   ]
   valreg'     = [(s,p) | (s,p) <- valreg,
                          s `elem` states'   ]
   rels'   = [(ag,x,y) | (ag,x,y) <- rels,
                           x `elem` states',
                           y `elem` states' ]
   actual' = [ s | s <- actual, s `elem` states' ]
```

Examples

```
m0 = initM [a,b,c] [P 0,Q 0]
```

Conversion of States to Integers Convert *any type of state list* to `[0..]`:

```
convert :: Eq state =>
           EpistM state -> EpistM Integer
convert (Mo states agents props regs
         valprop valreg rels actual) =
 Mo states' agents props regs valprop' valreg' rels' actual'
   where
    states' = map f states
    valprop'    = map (\ (x,y) -> (f x,y)) valprop
    valreg' = map (\ (x,y) -> (f x,y)) valreg
    rels'   = map (\ (x,y,z) -> (x, f y, f z)) rels
    actual' = map f actual
    f       = apply (zip states [0..])
```

Generated Submodels

```
gsm :: Ord state => EpistM state ->  EpistM state
gsm (Mo states ags props regs valprop valreg rel points) =
  (Mo states' ags props regs valprop' valreg' rel' points)
   where
   states' = closure rel ags points
   valprop'    = [(s,props) | (s,props)     <- valprop,
                              elem s states'              ]
   valreg'    = [(s,regs) | (s,regs)     <- valreg,
                              elem s states'              ]
   rel'    = [(ag,s,s') | (ag,s,s') <- rel,
                              elem s states',
                              elem s' states'             ]
```

The closure of a state list, given a relation and a list of agents:

```
closure ::  Ord state =>
            [(Agent,state,state)] ->
            [Agent] -> [state] -> [state]
closure rel agents xs = lfp f xs
 where f = \ ys -> (nub.sort) (ys ++ (expand rel agents ys))
```

The expansion of a relation R given a state set S and a set of agents B is given by $\{t \mid s \xrightarrow{b} t \in R, s \in S, b \in B\}$.

```
expand :: Ord state =>
           [(Agent,state,state)] ->
           [Agent] -> [state] -> [state]
expand rel agents ys = (nub . sort . concat)
  [ alternatives rel ag state | ag    <- agents,
                                state <- ys       ]
```

The epistemic alternatives for agent a in state s are the states in sR_a (the states reachable through R_a from s):

```
alternatives :: Eq state =>
     [(Agent,state,state)] ->
       Agent -> state -> [state]
alternatives rel ag current =
  [ s' | (a,s,s') <- rel, a == ag, s == current ]
```

Bisimulation: we compute the maximal bisimulation relation on an epistemic model by means of partition refinement.

Partition Refinement Given: A Kripke model $\mathbf{M}$.

Problem: find the Kripke model that results from replacing each state s in $\mathbf{M}$ by its bisimilarity class $|s|_{\leftrightarrow}$.

The problem of finding the smallest Kripke model modulo bisimulation is similar to the problem of minimizing the number of states in a finite automaton [J.E71].

We will use partition refinement, in the spirit of [PT87].

Partition Refinement Algorithm

- Start out with a partition of the state set where all states with the same valuation are in the same class.
- Given a partition Π, for each block b in Π, partition b into sub-blocks such that two states s, t of b are in the same sub-block iff for all agents a it holds that s and t have $\xrightarrow{a}$ transitions to states in the same block of Π. Update Π to Π' by replacing each b in Π by the newly found set of sub-blocks for b.
- Halt as soon as $\Pi = \Pi'$.

```
type State = Integer
```

Valuation Comparison

```
sameVal :: (Eq a,Eq b) => [(a,b)] -> a -> a -> Bool
sameVal val w1 w2 = apply val w1 == apply val w2
```

—- NOTE: updated until here

From Equivalence Relations to Partitions Relations as characteristic functions.

```
cf2part :: (Eq a) =>
             [a] -> (a -> a -> Bool) -> [[a]]
cf2part [] r = []
cf2part (x:xs) r = xblock : cf2part rest r
  where
  (xblock,rest) = (x : filter (r x) xs,
                   filter (not . (r x)) xs)
```

Initial Partition We start with the partition based on the relation 'having the same valuation':

```
initPartition :: Eq a => EpistM a -> [[a]]
initPartition (Mo states _ _ _ valprop valreg _ _) =
  cf2part states
   (\ x y -> (sameVal valprop x y && sameVal valreg x y))
```

The block of an object in a partition The block of `x` in a partition is the block that has `x` as an element.

```
bl :: Eq a => [[a]] -> a -> [a]
bl part x = head (filter (elem x) part)
```

Accessible Blocks For an agent from a given state, given a model and a partition:

```
accBlocks :: Eq a =>
           EpistM a -> [[a]] -> a -> Agent -> [[a]]
accBlocks m@(Mo _ _ _ _ _ _ rel _) part s ag =
    nub [ bl part y | (ag',x,y) <- rel,
                      ag' == ag, x == s ]
```

Having the same accessible blocks under a partition

```
sameAB :: Ord a =>
          EpistM a -> [[a]] -> a -> a -> Bool
sameAB m@(Mo _ ags _ _ _ _ _ _) part s t =
    and [ sort (accBlocks m part s ag)
          == sort (accBlocks m part t  ag) | ag <- ags ]
```

Refinement Step of Partition by Block Splitting Splitting the blocks `bl` of `p`:

```
refineStep :: Ord a => EpistM a -> [[a]] -> [[a]]
refineStep m p = refineP m p p
  where
  refineP :: Ord a =>
             EpistM a -> [[a]] -> [[a]] -> [[a]]
  refineP m part [] = []
  refineP m part (bl:blocks) =
     newblocks ++ (refineP m part blocks)
       where
         newblocks =
          cf2part bl (\ x y -> sameAB m part x y)
```

Refining a Partition The refining process can be implemented as a least fixpoint computation on the operation of taking refinement steps.

```
refine :: Ord a => EpistM a -> [[a]] -> [[a]]
refine m = lfp (refineStep m)
```

Remark: least fixpoint computation is an element of many refinement processes.

It is an example of what is called a *design pattern* in Software Engineering [GHJV95].

Construction of Minimal Model

```
minimalModel :: Ord a => EpistM a -> EpistM [a]
minimalModel m@(Mo states agents props regs
                valprop valreg rel actual) =
  (Mo states' agents props regs
   valprop' valreg' rel' actual')
     where
     states'   = refine m (initPartition m)
     f         = bl states'
     valprop'      = (nub . sort)
        (map (\ (x,y)   -> (f x, y))      valprop)
     valreg'      = (nub . sort)
        (map (\ (x,y)   -> (f x, y))      valreg)
     rel'      = (nub . sort)
        (map (\ (x,y,z) -> (x, f y, f z)) rel)
     actual'   = map f actual
```

Example:

```
*ModelsVocab> displayS5 s5example
[0,1,2,3]
[p,q]
```

```
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1,2,3]])
[1]

*ModelsVocab> displayS5 $ minimalModel s5example
[[0],[1],[2],[3]]
[p,q]
[([0],[]),([1],[p]),([2],[q]),([3],[p,q])]
(a,[[[0]],[[1]],[[2]],[[3]]])
(b,[[[0]],[[1]],[[2]],[[3]]])
(c,[[[0],[1],[2],[3]]])
[[1]]
```

Map to Bisimulation Minimal Model Map the states to their bisimilarity classes.

Next, convert the bisimilarity classes back into integers:

```
bisim ::  Ord a => EpistM a -> EpistM State
bisim = convert . minimalModel . gsm
```

Chapter 2

Action Models, Update

```
module ActionVocab where

import List
import ModelsVocab
import Test.QuickCheck
```

Definition of Action Models Datatype for Action Models. No need to specify a vocabulary, for the vocabulary is implicitly given by the list of precondition formulas.

```
data AM state = Am
              [state]
              [Agent]
              [(state,Form)]
              [(Agent,state,state)]
              [state]  deriving (Eq,Show)
```

Extracting the list of preconditions from an action model:

```
preconditions :: AM state -> [Form]
preconditions (Am _ _ pairs _ _) = map snd pairs
```

Extracting the proposition vocabulary from an action model:

```
vocProp :: AM state -> [Prp]
vocProp am = (sort.nub.concat)(map getPs (preconditions am))
```

Extracting the register vocabulary from an action model:

```
vocReg :: AM state -> [Reg]
vocReg am = (sort.nub.concat) (map getRs (preconditions am))
```

Functions from agent lists to action models

```
type FAM state = [Agent] -> AM state
```

Updating with an Action Model

```
up :: (Eq state, Ord state) =>
       EpistM state -> FAM state
                    -> EpistM (state,state)
```

```
up m@(Mo worlds ags props regs
      valprop valreg rel points) fam =
   Mo worlds' ags' props regs valprop' valreg' rel' points'
   where
   Am states ags' pre susp actuals = fam ags
   worlds' = [ (w,s) | w <- worlds, s <- states,
                   isTrueAtMayb m w (apply pre s) ==
                   Just True]
   valprop'    = [ ((w,s),props) | (w,props) <- valprop,
                                   s        <- states,
                                   elem (w,s) worlds']
   valreg'     = [ ((w,s),regs) | (w,regs) <- valreg,
                                   s        <- states,
                                   elem (w,s) worlds']
   rel'    = [ (ag1,(w1,s1),(w2,s2)) |
                 (ag1,w1,w2) <- rel,
                 (ag2,s1,s2) <- susp,
                  ag1 == ag2,
                  elem (w1,s1) worlds',
                  elem (w2,s2) worlds'   ]
   points' = [ (p,a) | p <- points, a <- actuals,
                       elem (p,a) worlds'         ]
```

Update and simplify

```
upd :: (Eq state, Ord state) =>
       EpistM state -> FAM state
                    -> EpistM State
upd m a = bisim (up m a)
```

Public Announcement Again Update model consists of a single action, with reflexive arrows for all agents.

Precondition is the formula that expresses the content of the announcement.

```
public :: Form -> FAM State
public form ags = Am [0] ags [(0,form)]
                      [(a,0,0)| a <- ags ] [0]
```

Composing Two Static Models

Definition of [EWS10].

```
composMod :: (Eq a, Ord a) =>
       EpistM a -> EpistM a
                    -> EpistM (a,a)

composMod m1@(Mo worlds agents props regs
                valprop1 valreg1 rel1 points)
          m2@(Mo worlds' agents' props' regs'
                valprop2 valreg2 rel2 points') =
              (Mo compstat agents compprops compregs
              compvalprop compvalreg comprel compoints) where

 compstat = [(x,y) | x <- worlds, y <- worlds',
             intersect (valPropStat x m1) (vcbPropSet m2) ==
             intersect (valPropStat y m2) (vcbPropSet m1),
             (map (apply (apply valreg1 x))
                  (intersect regs regs')) ==
             (map (apply (apply valreg2 y))
                  (intersect regs regs'))]

 comprel = [(i,(x,y),(r,s))| (i,x,r) <- rel1,
            (j,y,s)  <- rel2, i==j]

 compvalprop = [((x,y),
                 nub ((++) (vcbPropSet m1) (vcbPropSet m2)))
                | x <- worlds, y <- worlds']
 compvalreg = [((x,y),
                nub ((++) (valRegSet m1) (valRegSet m2)))
               | x <- worlds, y <- worlds']
 compprops = (sort.nub) ((++) props props')
 compregs = (sort.nub) ((++) regs regs')
 compoints = [(x,y) | x <- points, y <- points']
```

Compressing a Parallel Composition of two Models by Bisimulation

```
compos :: (Eq a, Ord a) =>
       EpistM a -> EpistM a -> EpistM State
compos m1 m2 = bisim (composMod m1 m2)
```

Chapter 3

Adding Factual Change

```
module ChangeVocab where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
```

Change in the World Following [BvEK06], we represent changes in the world as substitutions. A substitution maps proposition letters to formulas and registers to arithmetic expressions. Type of a substitition in Haskell:

```
type Subst = ([(Prp,Form)],[(Reg,Arith)])
```

Action+Change models

```
data ACM state = Acm
             [state]
             [Agent]
             [(state,(Form,Subst))]
             [(Agent,state,state)]
             [state]  deriving (Eq,Show)
```

Extracting the list of preconditions from an action model:

```
preconditions :: ACM state -> [Form]
preconditions (Acm _ _ pairs _ _) = map (fst.snd) pairs
```

Extracting the proposition vocabulary from an action model:

```
vocProp :: ACM state -> [Prp]
vocProp acm@(Acm _ _ pairs _ _) = (sort.nub)
  ((concat (map getPs (preconditions acm))) ++
   (map fst propsubs)               ++
   (concat (map (getPs.snd) propsubs)))
 where propsubs = concat (map (fst.snd.snd) pairs)
```

Extracting the register vocabulary from an action model:

```
vocReg :: ACM state -> [Reg]
vocReg acm@(Acm _ _ pairs _ _) = (sort.nub)
  ((concat (map getRs (preconditions acm))) ++
   (map fst regsubs)               ++
   (concat (map (getRsA.snd) regsubs)))
 where regsubs = concat (map (snd.snd.snd) pairs)
```

Functions from Agents to A+C models

```
type FACM state = [Agent] -> ACM state
```

Getting the precondition and the substitution from an A+C model

```
prec :: ACM state -> [(state,Form)]
prec (Acm _ _ ps _ _) =
    map (\ (x,(y,_)) -> (x,y)) ps

subst :: ACM state -> [(state,Subst)]
subst (Acm _ _ ps _ _) =
    map (\ (x,(_,y)) -> (x,y)) ps
```

From Tables to Functions

```
t2f :: Eq a => [(a,b)] -> a -> b
t2f t = \ x -> maybe undefined id (lookup x t)
```

Extracting the proposition and register substitutions as functions:

```
subProp :: Eq a => ACM a -> a -> Prp -> Form
subProp am s p = let
     sb = fst (t2f (subst am) s) in
  if elem p (map (\ (x,_) -> x) sb) then
     t2f sb p
   else (Prp p)
```

```
subReg :: Eq a => ACM a -> a -> Reg -> Arith
subReg am s r = let
     sb = snd (t2f (subst am) s) in
  if elem r (map (\ (x,_) -> x) sb) then
     t2f sb r
   else (Reg r)
```

Changing the World Valuation at a world in an epistemic model:

```
valProp :: Eq a => EpistM a -> a -> [Prp]
valProp m = t2f (valuationProp m)
```

```
valReg :: Eq a => EpistM a -> a -> [(Reg,Int)]
valReg m = t2f (valuationReg m)
```

New valuation after update with an action model

```
newValProp :: (Eq a, Ord a) =>
  EpistM a -> ACM a -> (a,a) -> [Prp]
newValProp m am (w,s) = [ p | p <- allprops, subfct p ]
   where
   allprops = (sort.nub)
    ((valProp  m w) ++
     (map (\ (x,_) -> x) (fst (t2f (subst am) s))))
   subfct p = isTrueAtMayb m w (subProp am s p) == Just True
```

```
newValReg :: (Eq a, Ord a) =>
  EpistM a -> ACM a -> (a,a) -> [(Reg,Int)]
newValReg m am (w,s) = [ (r,arithVal m w (subReg am s r))
    | r <- vocabReg m]
```

Updating with an A+C Model

```
upc :: (Eq state, Ord state) =>
       EpistM state -> FACM state
                    -> EpistM (state,state)
```

```
upc m@(Mo worlds ags props regs
       valprop valreg rel points) facm =
   Mo worlds' ags' props' regs' valprop' valreg' rel' points'
   where
    acm@(Acm states ags' ps susp as) = facm ags
    worlds' = [ (w,s) | w <- worlds, s <- states,
                isTrueAtMayb m w (apply (prec acm) s)
                == Just True ]
    props' = (sort.nub) (props ++ vocProp acm)
    regs' = (sort.nub) (regs ++ vocReg acm)
    valprop'   = [ ((w,s),newValProp m acm (w,s)) |
                                  (w,s) <- worlds' ]
    valreg'   = [ ((w,s),newValReg m acm (w,s)) |
                                  (w,s) <- worlds' ]
    rel'    = [ (ag1,(w1,s1),(w2,s2)) |
                (ag1,w1,w2) <- rel,
                (ag2,s1,s2) <- susp,
                 ag1 == ag2,
                 elem (w1,s1) worlds',
                 elem (w2,s2) worlds'   ]
    points' = [ (p,a) | p <- points, a <- as,
                        elem (p,a) worlds'    ]
```

Update and Simplify

```
upd ::  (Eq state, Ord state) =>
       EpistM state -> FACM state
                    -> EpistM State
upd m a = bisim (upc m a)
```

String a series of updates together:

```
upds :: EpistM State -> [FACM State] -> EpistM State
upds m [] = m
upds m (a:as) = upds (upd m a) as
```

Public Announcement See [Pla89, Ger99].

Update model consists of a single action, with reflexive arrows for all agents.

Precondition is the formula that expresses the content of the announcement.

```
public :: Form -> FACM State
public form ags = Acm [0] ags [(0,(form,([],[])))]
                      [(a,0,0)| a <- ags ] [0]
```

Example

```
m0 = upc s5example (public p)
m1 = upd s5example (public p)
```

```
*ChangeVocab> displayS5 s5example
[0,1,2,3]
[p,q]
[R0]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
[(0,[(R0,0)]),(1,[(R0,1)]),(2,[(R0,2)]),(3,[(R0,3)])]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1,2,3]])
[1]
```

```
*ChangeVocab> displayS5 m0
[(1,0),(3,0)]
[p,q]
[R0]
[((1,0),[p]),((3,0),[p,q])]
[((1,0),[(R0,1)]),((3,0),[(R0,3)])]
(a,[[(1,0)],[(3,0)]])
(b,[[(1,0)],[(3,0)]])
(c,[[(1,0),(3,0)]])
[(1,0)]


*ChangeVocab> displayS5 m1
[0,1]
[p,q]
[R0]
[(0,[p]),(1,[p,q])]
[(0,[(R0,1)]),(1,[(R0,3)])]
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
[0]
```

Public Change

```
pChange :: Subst -> FACM State
pChange subst ags = Acm [0] ags [(0,(Top,subst))]
                      [(a,0,0)| a <- ags ] [0]
```

Example

```
m2 = upc s5example (pChange ([(P 0,q)],[(Rg 0,I 0)]))
m3 = upd s5example (pChange ([(P 0,q)],[(Rg 0,I 0)]))
```

```
*ChangeVocab> displayS5 m2
[(0,0),(1,0),(2,0),(3,0)]
[p,q]
[R0]
[((0,0),[]),((1,0),[]),((2,0),[p,q]),((3,0),[p,q])]
[((0,0),[(R0,0)]),((1,0),[(R0,0)]),((2,0),[(R0,0)]),((3,0),[(R0,0)])]
(a,[[(0,0)],[(1,0)],[(2,0)],[(3,0)]])
(b,[[(0,0)],[(1,0)],[(2,0)],[(3,0)]])
(c,[[(0,0),(1,0),(2,0),(3,0)]])
[(1,0)]

*ChangeVocab> displayS5 m3
[0,1]
[p,q]
[R0]
[(0,[]),(1,[p,q])]
[(0,[(R0,0)]),(1,[(R0,0)])]
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
[0]
```

Group Announcement Computing the update for passing a group announcement to a list of agents: the other agents confuse the action with the action where nothing happens.

In the limit case where the message is passed to all agents, the message is a public announcement.

```
groupM :: [Agent] -> Form -> FACM State
groupM gr form agents =
  if sort gr == sort agents
    then public form agents
    else
      (Acm
       [0,1]
        agents
       [(0,(form,([],[]))),(1,(Top,([],[])))]
        ([ (a,0,0) | a <- agents ]
         ++ [ (a,0,1) | a <- agents \\ gr ]
         ++ [ (a,1,0) | a <- agents \\ gr ]
         ++ [ (a,1,1) | a <- agents          ])
       [0])
```

Example

```
e0 = initM [a,b,c] [P 0,Q 0]
m4 = upc e0 (groupM [a,b] (Neg p))
m5 = upd e0 (groupM [a,b] (Neg p))
```

```
*ChangeVocab> displayS5 e0
[0,1,2,3]
[p,q]
[]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
[(0,[]),(1,[]),(2,[]),(3,[])]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
[0,1,2,3]

*ChangeVocab> displayS5 m4
```

```
[(0,0),(0,1),(1,1),(2,0),(2,1),(3,1)]
[p,q]
[]
[((0,0),[]),((0,1),[]),((1,1),[p]),((2,0),[q]),((2,1),[q]),((3,1),[p,q])]
[((0,0),[]),((0,1),[]),((1,1),[]),((2,0),[]),((2,1),[]),((3,1),[])]
(a,[[(0,0),(2,0)],[(0,1),(1,1),(2,1),(3,1)]])
(b,[[(0,0),(2,0)],[(0,1),(1,1),(2,1),(3,1)]])
(c,[[(0,0),(0,1),(1,1),(2,0),(2,1),(3,1)]])
[(0,0),(2,0)]

*ChangeVocab> displayS5 m5
[0,1,2,3,4,5]
[p,q]
[]
[(0,[]),(1,[]),(2,[p]),(3,[q]),(4,[q]),(5,[p,q])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[])]
(a,[[0,3],[1,2,4,5]])
(b,[[0,3],[1,2,4,5]])
(c,[[0,1,2,3,4,5]])
[0,3]
```

Private Messages Private messages are a special case of group messages:

```
message :: Agent -> Form -> FACM State
message agent = groupM [agent]
```

Tests Tests are another special case of group messages:

```
test :: Form -> FACM State
test = groupM []
```

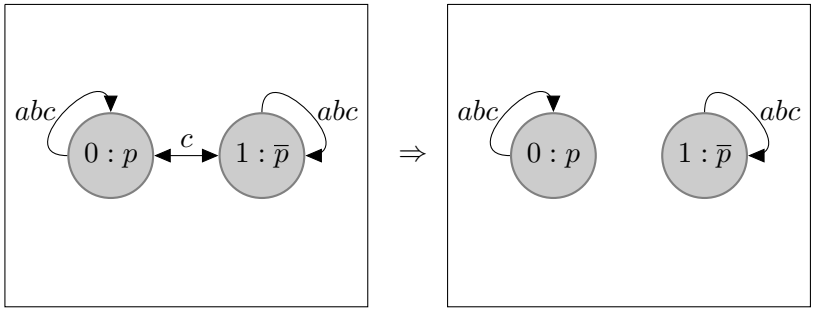
Communications Whether

- informing everone *whether* φ,
- informing a group *whether* φ,
- informing an individual *whether* φ.

Telling someone whether it rains involves giving her the facts: if it rains you tell her "it rains", if it does not rain you tell her "it does not rain".

In the action model for this there are *two* actual actions. Which one will lead to a new actual world depends on the facts of the matter, and these are determined by the input model ...

General Form: Group Communication Whether Informing Everyone Whether p.



Implementation First the negation of a formula:

```
negation :: Form -> Form
negation (Neg form) = form
negation form       = (Neg form)
```

Informing a Group Whether φ

```
info :: [Agent] -> Form -> FACM State
info group form agents =
   Acm
   [0,1]
   agents
   [(0,(form,([],[]))),(1,(negation form,([],[])))]
   ([ (a,0,0) | a <- agents ]
     ++ [ (a,1,1) | a <- agents ]
     ++ [ (a,0,1) | a <- others ]
     ++ [ (a,1,0) | a <- others ])
   [0,1]
     where others = agents \\ group
```

Example

```
m6 = upc e0 (info [a,b] p)
m7 = upd e0 (info [a,b] p)
```

```
*ChangeVocab> displayS5 m6
[(0,1),(1,0),(2,1),(3,0)]
[p,q]
[]
[((0,1),[]),((1,0),[p]),((2,1),[q]),((3,0),[p,q])]
[((0,1),[]),((1,0),[]),((2,1),[]),((3,0),[])]
(a,[[(0,1),(2,1)],[(1,0),(3,0)]])
(b,[[(0,1),(2,1)],[(1,0),(3,0)]])
(c,[[(0,1),(1,0),(2,1),(3,0)]])
[(0,1),(1,0),(2,1),(3,0)]

*ChangeVocab> displayS5 m7
```

```
[0,1,2,3]
[p,q]
[]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
[(0,[]),(1,[]),(2,[]),(3,[])]
(a,[[0,2],[1,3]])
(b,[[0,2],[1,3]])
(c,[[0,1,2,3]])
[0,1,2,3]
```

Chapter 4

Change and Perception

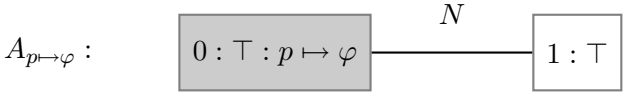
Based on [Eijar].

```
module ChangePerception where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
import ChangeVocab
```

Unobserved Change

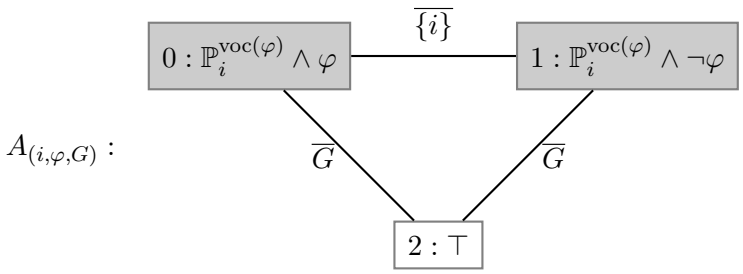
The action model $A_{p:=\varphi}$ for unobserved change $p := \varphi$ looks as follows (note that reflexive arrows are not drawn, and it is assumed that N is the set of all agents):



Implementation:

```
unobserved_change :: Prp -> Form -> FACM State
unobserved_change prp form ags =
  Acm
   [0,1]
   ags
   [(0,(Top,([(prp,form)],[]))), (1,(Top,([],[])))]
   [(a,s,t)| a <- ags, s <- [0,1], t <- [0,1]]
   [0]
```

The action model $A_{(i,\varphi,G)}$ for a perception by i of φ, witnessed by G, looks as follows (if Q is a list of proposition letters, $\mathbb{P}_i^Q$ expresses that i is able to perceive the letters in Q; $\mathrm{voc}(\varphi)$ gives the propositional vocabulary of formula φ):



Implementation (assuming for simplicity that every agent can perceive every proposition letter):

```
perception :: Agent -> Form -> [Agent] -> FACM State
perception i form group ags =
  Acm [0,1,2]
       ags
      [(0,(form,([],[]))),
       (1,(Neg form, ([],[]))),
       (2,(Top,([],[])))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

First model:

```
md0 :: EpistM Integer
md0 = Mo [0,1]
         [a,b,c]
         [P 0]
         []
         [(0,[P 0]), (1,[])]
         [(0,[]), (1,[])]
         ([(ag,x,x) | ag <- [a,b,c], x <- [0,1]]
               ++ [(c,0,1),(c,1,0)])
         [0]
```

Second model:

```
md1 = upd md0 (perception b (Prp (P 0)) [b,c])
md1' = upc md0 (perception b (Prp (P 0)) [b,c])
```

Third model:

```
md2 = upd md1 (unobserved_change (P 0) (Neg Top))
md2' = upc md1 (unobserved_change (P 0) (Neg Top))
```

Fourth model:

```
md3 = upd md2 (perception b (Neg p) [b,c])
md3' = upc md2 (perception b (Neg p) [b,c])
```

Fourth model, after perception by all:

```
md3a = upd md2 (perception b (Neg p) [a,b,c])
md3a' = upc md2 (perception b (Neg p) [a,b,c])
```

Different update:

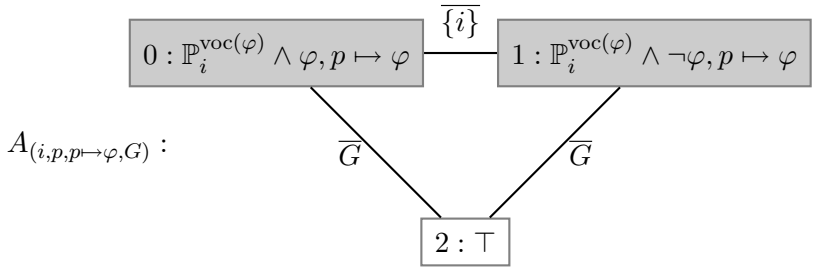
```
md1a = upd md0 (perception c p [a,c])
md1a' = upc md0 (perception c p [a,c])
```

Different initial model:

```
mm0 :: EpistM Integer
mm0 = Mo [0,1]
         [a,b,c]
         [P 0]
         []
         [(0,[P 0]), (1,[])]
         [(0,[]),(1,[])]
         ([(ag,x,x) | ag <- [a,b,c], x <- [0,1]]
            ++ [(a,0,1),(a,1,0)]
            ++ [(c,0,1),(c,1,0)])
            [0]
```

```
mm1 = upd md0 (perception b (Prp (P 0)) [b,c])
mm1' = upc md0 (perception b (Prp (P 0)) [b,c])
```

The action model for perceived change $(i, p, p \mapsto \varphi, G)$ (perception by i of p after a change in p has taken place in the model, with G as witnesses of the act of perception) takes the following shape:



Implementation:

```
perceived_change ::  Agent ->
                       Prp -> Form -> [Agent] -> FACM State
perceived_change i prp form group ags =
  Acm [0,1,2] ags [(0,(Prp prp,([(prp,form)],[]))),
                   (1,(Neg (Prp prp), ([(prp,form)],[]))),
                   (2,(Top,([],[])))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

Some updates with this:

```
me1  = upd md0 (perceived_change a (P 0) (Top) [a])
me1' = upc md0 (perceived_change a (P 0) (Top) [a])
```

```
me2  = upd md0 (perceived_change a (P 0) (Neg Top) [a])
me2' = upc md0 (perceived_change a (P 0) (Neg Top) [a])
```

```
me3  = upd md0 (perceived_change a (P 0) (Neg Top) [a,b])
me3' = upc md0 (perceived_change a (P 0) (Neg Top) [a,b])
```

```
me4  = upd md1 (perceived_change a (P 0) (Neg Top) [a,b])
me4' = upc md1 (perceived_change a (P 0) (Neg Top) [a,b])
```

```
ppc ::  Agent -> Prp -> Form -> [Agent] -> FACM State
ppc i prp form group ags =
  Acm [0,1,2] ags [(0,(form,([(prp,form)],[]))),
                   (1,(Neg form, ([(prp,form)],[]))),
                   (2,(Top,([],[])))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

```
npc ::  Agent -> Prp -> Form -> [Agent] -> FACM State
npc i prp form group ags =
  Acm [0,1,2] ags [(0,(form,([(prp,form)],[]))),
                   (1,(negation form, ([(prp,form)],[]))),
                   (2,(Top,([],[])))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [1]
```

And again:

```
mpc1  = upd md0 (ppc a (P 0) (Top) [a])
mpc1' = upc md0 (ppc a (P 0) (Top) [a])
```

```
mpc2  = upd md0 (npc a (P 0) (Neg Top) [a])
mpc2' = upc md0 (npc a (P 0) (Neg Top) [a])
```

```
mpc3  = upd md0 (npc a (P 0) (Neg Top) [a,b])
mpc3' = upc md0 (npc a (P 0) (Neg Top) [a,b])
```

```
mpc4  = upd md1 (npc a (P 0) (Neg Top) [a,b])
mpc4' = upc md1 (npc a (P 0) (Neg Top) [a,b])
```

Chapter 5

The Muddy Children Puzzle

```
module Muddy where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
import ChangeVocab
import ChangePerception
```

Abbreviations for some basic propositions:

```
ma, mb, mc, md :: Form
ma = Prp (P 1) -- this represents Alice is muddy
mb = Prp (P 2) -- this represents Bob is muddy
mc = Prp (P 3) -- this represents Carol is muddy
md = Prp (P 4) -- this represents Dave is muddy
```

Let's model the case where Bob, Carol and Dave are muddy:

```
bcd_dirty = Conj [Neg ma, mb, mc, md]
```

The following series of updates expresses that each child is aware of the state (muddy or not) of the other children:

```
awareness = [info [b,c,d] ma,
             info [a,c,d] mb,
             info [a,b,d] mc,
             info [a,b,c] md ]
```

Formulas for knowing whether one is muddy:

```
aKn =  Disj [K a ma, K a (Neg ma)]
bKn =  Disj [K b mb, K b (Neg mb)]
cKn =  Disj [K c mc, K c (Neg mc)]
dKn =  Disj [K d md, K d (Neg md)]
```

We start with an initial situation where the four agents are blissfully unaware about the muddiness facts, and update with the test expressing that b,c,d are in fact muddy. This gives the following model:

```
mu0 = upd (initM [a,b,c,d] [P 1, P 2, P 3, P 4])
          (test bcd_dirty)
```

Next, add awareness information:

```
mu1 = upds mu0 awareness
```

This gives:

```
*Muddy> displayS5 mu1
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[p1,p2,p3,p4]
[]
[(0,[]),(1,[p1]),(2,[p2]),(3,[p3]),(4,[p4]),(5,[p1,p2]),(6,[p1,p3]),
 (7,[p1,p4]),(8,[p2,p3]),(9,[p2,p4]),(10,[p3,p4]),(11,[p1,p2,p3]),
 (12,[p1,p2,p4]),(13,[p1,p3,p4]),(14,[p2,p3,p4]),(15,[p1,p2,p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[]),(6,[]),(7,[]),(8,[]),
 (9,[]),(10,[]),(11,[]),(12,[]),(13,[]),(14,[]),(15,[])]
(a,[[0,1],[2,5],[3,6],[4,7],[8,11],[9,12],[10,13],[14,15]])
(b,[[0,2],[1,5],[3,8],[4,9],[6,11],[7,12],[10,14],[13,15]])
(c,[[0,3],[1,6],[2,8],[4,10],[5,11],[7,13],[9,14],[12,15]])
(d,[[0,4],[1,7],[2,9],[3,10],[5,12],[6,13],[8,14],[11,15]])
[14]
```

Update with a public announcement of the father that at least one child is muddy.

```
mu2 = upd  mu1 (public (Disj [ma, mb, mc, md]))
```

This gives:

```
*Muddy> displayS5 mu2
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
[p1,p2,p3,p4]
[]
[(0,[p1]),(1,[p2]),(2,[p3]),(3,[p4]),(4,[p1,p2]),(5,[p1,p3]),(6,[p1,p4]),
(7,[p2,p3]),(8,[p2,p4]),(9,[p3,p4]),(10,[p1,p2,p3]),(11,[p1,p2,p4]),
(12,[p1,p3,p4]),(13,[p2,p3,p4]),(14,[p1,p2,p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[]),(6,[]),(7,[]),(8,[]),
 (9,[]),(10,[]),(11,[]),(12,[]),(13,[]),(14,[])]
(a,[[0],[1,4],[2,5],[3,6],[7,10],[8,11],[9,12],[13,14]])
(b,[[0,4],[1],[2,7],[3,8],[5,10],[6,11],[9,13],[12,14]])
(c,[[0,5],[1,7],[2],[3,9],[4,10],[6,12],[8,13],[11,14]])
(d,[[0,6],[1,8],[2,9],[3],[4,11],[5,12],[7,13],[10,14]])
[13]
```

The first round: they all say they don't know their state.

```
mu3 = upd  mu2
       (public (Conj[Neg aKn, Neg bKn, Neg cKn, Neg dKn]))
```

```
*Muddy> displayS5 mu3
[0,1,2,3,4,5,6,7,8,9,10]
[p1,p2,p3,p4]
[]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),(3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4]),
 (6,[p1,p2,p3]),(7,[p1,p2,p4]),(8,[p1,p3,p4]),(9,[p2,p3,p4]),(10,[p1,p2,p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[]),(6,[]),(7,[]),(8,[]),(9,[]),(10,[])]
(a,[[0],[1],[2],[3,6],[4,7],[5,8],[9,10]])
(b,[[0],[1,6],[2,7],[3],[4],[5,9],[8,10]])
(c,[[0,6],[1],[2,8],[3],[4,9],[5],[7,10]])
(d,[[0,7],[1,8],[2],[3,9],[4],[5],[6,10]])
[9]
```

The second round: they still all don't know their state.

```
mu4 = upd  mu3
       (public (Conj[Neg aKn, Neg bKn, Neg cKn, Neg dKn]))
```

```
*Muddy> displayS5 mu4
[0,1,2,3,4]
[p1,p2,p3,p4]
[]
[(0,[p1,p2,p3]),(1,[p1,p2,p4]),(2,[p1,p3,p4]),(3,[p2,p3,p4]),(4,[p1,p2,p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[])]
(a,[[0],[1],[2],[3,4]])
(b,[[0],[1],[2,4],[3]])
(c,[[0],[1,4],[2],[3]])
(d,[[0,4],[1],[2],[3]])
[3]
```

Now b, c and d say they know. In the final model all is known to everyone.

```
mu5 = upds mu4 [public (Conj[bKn, cKn, dKn])]
```

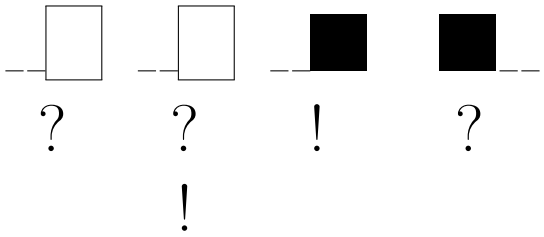
```
*Muddy> displayS5 mu5
[0]
[p1,p2,p3,p4]
[]
[(0,[p2,p3,p4])]
[(0,[])]
(a,[[0]])
(b,[[0]])
(c,[[0]])
(d,[[0]])
[0]
```

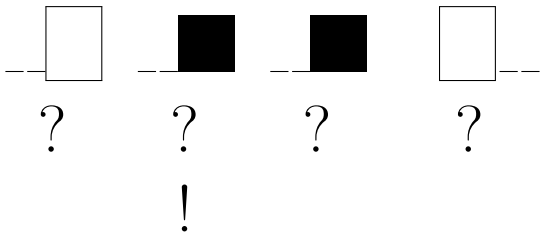
Chapter 6

The Wise Men Puzzle; or: The Riddle of the Caps

```
module WiseMen

where
import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
import ChangeVocab
import ChangePerception
```





Analysis with Model Checking

- Four agents: a, b, c, d, occupying positions $1, 2, 3, 4$.
- Four basic propositions p_1, p_2, p_3, p_4.
- p_i expresses that the guy at position i is wearing a *white* cap.

Initial model

```
mo0 = initM [a,b,c,d] [P 1, P 2, P 3, P 4]
```

```
p1,p2,p3,p4 :: Form
p1 = Prp (P 1); p2 = Prp (P 2)
p3 = Prp (P 3); p4 = Prp (P 4)

capsInfo :: Form
capsInfo =
  Disj [Conj [f, g, Neg h, Neg j] |
              f <- [p1, p2, p3, p4],
              g <- [p1, p2, p3, p4] \\ [f],
              h <- [p1, p2, p3, p4] \\ [f,g],
              j <- [p1, p2, p3, p4] \\ [f,g,h],
              f < g, h < j                    ]
```

```
mo1 = upd mo0 (public capsInfo)
```

```
*WiseMen> displayS5 mo1
[0,1,2,3,4,5]
[p1,p2,p3,p4]
[]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),
 (3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[])]
(a,[[0,1,2,3,4,5]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
(d,[[0,1,2,3,4,5]])
[0,1,2,3,4,5]
```

```
awarenessFirstCap  = info [b,c] p1
awarenessSecondCap = info [c]   p2

mo2  = upd (upd mo1 awarenessFirstCap)
                    awarenessSecondCap
```

```
*WiseMen> displayS5 mo2
[0,1,2,3,4,5]
[p1,p2,p3,p4]
[]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),
 (3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4])]
[(0,[]),(1,[]),(2,[]),(3,[]),(4,[]),(5,[])]
(a,[[0,1,2,3,4,5]])
(b,[[0,1,2],[3,4,5]])
(c,[[0],[1,2],[3,4],[5]])
```

```
(d,[[0,1,2,3,4,5]])
[0,1,2,3,4,5]
```

```
bK =  Disj [K b p2, K b (Neg p2)]
cK =  Disj [K c p3, K c (Neg p3)]

mo3a = upd  mo2 (public cK)
mo3b = upd  mo2 (public (Neg cK))
```

```
*WiseMen> displayS5 mo3a
[0,1]
[p1,p2,p3,p4]
[]
[(0,[p1,p2]),(1,[p3,p4])]
[(0,[]),(1,[])]
(a,[[0,1]])
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0,1]])
[0,1]

*WiseMen> displayS5 mo3b
[0,1,2,3]
[p1,p2,p3,p4]
[]
[(0,[p1,p3]),(1,[p1,p4]),(2,[p2,p3]),(3,[p2,p4])]
[(0,[]),(1,[]),(2,[]),(3,[])]
(a,[[0,1,2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1],[2,3]])
(d,[[0,1,2,3]])
[0,1,2,3]
```

```
impl :: Form -> Form -> Form
impl form1 form2 = Disj [Neg form1, form2]

equiv :: Form -> Form -> Form
equiv form1 form2 =
  Conj [form1 `impl` form2, form2 `impl` form1]
```

```
test1 = isTrue mo3a bK
test2 = isTrue mo3b bK
test3 = isTrue mo3a (K a (equiv p1 p2))
test4 = isTrue mo3b (K a (equiv p1 (Neg p2)))
```

```
*WiseMen> test1
Just True
*WiseMen> test2
Just True
*WiseMen> test3
Just True
*WiseMen> test4
Just True
```

```
mo4a = upd mo3a (public bK)
mo4b = upd mo3b (public bK)
```

```
*WiseMen> displayS5 mo4a
[0,1]
```

```
[p1,p2,p3,p4]
[]
[(0,[p1,p2]),(1,[p3,p4])]
[(0,[]),(1,[])]
(a,[[0,1]])
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0,1]])
[0,1]

*WiseMen> displayS5 mo4b
[0,1,2,3]
[p1,p2,p3,p4]
[]
[(0,[p1,p3]),(1,[p1,p4]),(2,[p2,p3]),(3,[p2,p4])]
[(0,[]),(1,[]),(2,[]),(3,[])]
(a,[[0,1,2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1],[2,3]])
(d,[[0,1,2,3]])
[0,1,2,3]
```

Chapter 7

Liar's Dice

In this section we show how the game of Liar's Dice which is analysed in [DvESW07] can be modelled using DEMO, and we demonstrate the doxastic models that we get if we trace a particular run of the game.

First we will closely examine the different actions that take place in the game and their representations as action models. Let p represent the value of a coin, with 1 signifying heads, and 0 signifying tails. Let agents a and b represent the two players, and let C_1 represent the contents of the purse of player a (C for cash), and C_2 that of player b, with natural number values representing the amounts in euros that each player has in her purse. These natural number registers are available in the new extension of DEMO. Let S_1, S_2 represent the money at stake for each player. Factual change can be thought of as assignment of new values to variables. This is an essential ingredient of the various actions in the game:

Initialisation Both players put one euro at stake, and they both know this. $S_1 := 1, C_1 := C_1 - 1, S_2 := 1, C_2 := C_2 - 1$, together with public announcement of these factual changes.

Heads Factual change of the propositional value of a coin p to 1, with private communication of the result to player a ($p = 1$ signifies heads).

Tails Factual change of the propositional value of a coin p to 0, with private communication of the result to player a. ($p = 0$ signifies tails).

Announce Player a announces either $\ddagger Head$ or $\ddagger Tail$. There are several ways to model this and we will come back to this later.

Pass Player b passes and loses, player a gets the stakes. $C_1 := C_1 + S_1 + S_2, S_1 := 0, S_2 := 0$.

Challenge Public setting of $C_2 := C_2 - 1, S_2 := S_2 + 1$, followed by public announcement of the value of p. If the outcome is p then $C_1 := C_1 + S_1 + S_2$, otherwise $C_2 := C_2 + S_1 + S_2$ and in any case $S_1 := 0, S_2 := 0$.

We will show how these actions can be defined as doxastic action models in Haskell code using DEMO.

```
module Lies
where
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,
                           vocProp,vocReg)
import ChangeVocab
import ChangePerception
import Data.Set (Set)
import qualified Data.Set as Set
```

```
type EM = EpistM Integer
```

We first define the cash and stakes of each player as integer registers.

```
c1, c2, s1, s2 :: Reg
c1 = (Rg 1); c2 = (Rg 2)
s1 = (Rg 3); s2 = (Rg 4)
```

This declares four integer registers, and gives them appropriate names. The initial contents of the purses of the two players must also be defined. Let's assume both players have five euros in cash to start with.

```
initCash1, initCash2 :: Int
initCash1 = 5
initCash2 = 5
```

Initialisation of the game: both players put one euro at stake. This is modelled by the following factual change: $S_1 := 1, C_1 := C_1 - 1, S_2 := 1, C_2 := C_2 - 1$. The representation of this in our modelling language is straightforward. We just represent the contents of the registers at startup.

```
initGame :: EM
initGame = (Mo
            [0]
            [a,b]
    []
    [s1, s2, c1, c2]
    [(0,[])]
            [(0,[(s1,1),(s2,1),
                 (c1,(initCash1-1)),(c2,(initCash2-1))])]
            [(a,0,0),(b,0,0)]
            [0])
```

Tossing the coin is a factual change of p to 0 or 1. The coin is tossed secretly and before player a looks both players don't know the value of the coin. Because of this there are two worlds, one where p is set to 0 and one where p is set to 1, and neither of the two players can distinguish these worlds.

```
toss :: Integer -> FACM State
toss c ags = (Acm
              [0,1]
              ags
              [(0,(Top,([(P 0,Neg Top)],[]))),
               (1,(Top,([(P 0,Top)],[])))]
              [(ag,w,w') | w  <- [0,1],
                           w' <- [0,1], ag <- ags]
              [c])
```

Note that the action model has a list that assigns to each world a precondition, a change to the propositions, and a change to the registers. In world 0, the precondition is $\top$ and the change is to set p to value $\neg\top$, i.e., $\bot$ (and there is no change to the registers), and in world 1, the precondition is again $\top$ and the change is to set p to value $\top$ (and again, there is no change to the registers).

After the coin is tossed player a looks under the cup without showing the coin to player b. We define a generic function for computing the model of the action where a group of agents looks under the cup. These models consist of two worlds, one where p is true (heads) and one where p is false (tails), the agents in the group can distinguish these two worlds and the other agents cannot.

```
look :: [Agent] -> FACM State
look group ags = (Acm
                  [0,1]
                  ags
                  [(0,(p,([],[]))),(1,(Neg(p),([],[])))]
                  ([(ag,w,w') | w <- [0,1], w' <- [0,1],
                                ag <- ags, notElem ag group] ++
                   [(ag,w,w)  | w <- [0,1], ag <- group])
                  [0,1])
```

In this case, there are no changes to propositions or registers, but world 0 has precondition p, and world 1 has precondition $\neg p$.

Now we define the models of the situation after the coin has been tossed and player a has looked at the outcome, distinguishing the two outcomes of the toss:

```
headsg :: EM
headsg = upd (upd initGame (toss 1)) (look [a])

tailsg :: EM
tailsg = upd (upd initGame (toss 0)) (look [a])
```

Before looking at the way to model the announcement of an outcome of the toss by player a we will first define the action models for passing and challenging.

When player b passes, the stakes are added to player a's cash: $C_2 := C_2 + S_1 + S_1, S_1 := 0, S_2 := 0$. Player b never gets to see the actual value of the coin so there are no changes in the knowledge of the agents about p. The model for this has only one world that indicates the changes in the stakes and cash.

```
pass :: FACM State
pass ags = (Acm
            [0]
            ags
            [(0,(Top,([],
                  [(s1,(I 0)),
                   (s2,(I 0)),
                   (c1,ASum [Reg c1,Reg s1,Reg s2])])))]
            [(ag,0,0) | ag <- ags]
            [0])
```

Note that here for the first time we see changes to the registers.

When player b decides to challenge player a, the cup is lifted and both players get to know the value of p. Then the stakes are added to the cash of player a in case of heads and player b in case of tails, together with one

extra euro from the cash of player b that player b added to the stakes while challenging player a. So instead of $S2 := S2 + 1, C2 := C2 - 1$ and after that $C1 := C1 + S1 + S2$ in case of heads and $C2 := C2 + S1 + S2$ in case of tails, we use $C1 := C1 + S1 + S2 + 1, C2 := C2 - 1$ in case of heads and $C2 := C2 + S1 + S2$ in case of tails. The action model for this has one world for the case of heads and one world for the case of tails. Both players can distinguish these worlds because the cup was lifted, and the stakes are divided differently in the two worlds.

```
challenge :: FACM State
challenge ags =
 Acm
 [0,1]
 ags
 [(0,(Neg(p),([],
       [(s1,(I 0)),
        (s2,(I 0)),
        (c2,ASum [Reg c2,Reg s1,Reg s2])]))),
  (1,(    p ,([],
       [(s1,(I 0)),
        (s2,(I 0)),
        (c2,ASum [Reg c2,I (-1)]),
        (c1,ASum [Reg c1,Reg s1,Reg s2,I 1])])))]
  [(ag,w,w) | w <- [0,1], ag <- ags]
  [0,1]
```

When player a announces $\ddagger Head$ or $\ddagger Tail$ the stakes change. In case of $\ddagger Head\, C_1 := C_1 - 1, S_1 := S_1 + 1$ and in case of $\ddagger Tail\, C_2 := C_2 + S_1 + S_2, S_1 := 0, S_2 := 0$.

```
announceStakes :: Integer -> FACM State
announceStakes 0 ags =
 Acm
 [0]
 ags
 [(0,(Top,([],[(s1,(I 0)),
  (s2,(I 0)),
  (c2,ASum [Reg c2,Reg s1,Reg s2])])))]
 [(ag,0,0) | ag <- ags]
 [0]
announceStakes 1 ags =
 Acm
 [0]
 ags
 [(0,(Top,([],[(s1,ASum [Reg s1,I 1]),
  (c1,ASum [Reg c1,I (-1)])])))]
 [(ag,0,0) | ag <- ags]
 [0]
```

Now the only thing we have to decide is how we will model the announcement of $\ddagger Head$ or $\ddagger Tail$. Suppose we would use the manipulative update $\ddagger p$ or $\ddagger \neg p$ for this. This would imply that the other player believes the claims that are made.

However, in a real game of Liar's Dice player b knows that player a might very well be bluffing and she doesn't really believe player a's claim at all. So to correctly model the game we should not use the manipulative update. When player a makes an announcement this doesn't even change player b's knowledge and beliefs because player b doesn't believe player a.

So instead of the manipulative update we should only use the model for changing the stakes to model the announcement:

```
announce :: Integer -> FACM State
announce = announceStakes
```

Now player b doesn't know whether p is true but she knows she doesn't

know:

```
bKnows :: Form
bKnows = Disj [(K b (Neg p)), (K b p)]
```

```
*Lies> isTrue (upd tailsg (announce 0)) bKnows
Just False
*Lies> isTrue (upd tailsg (announce 0)) (K b (Neg bKnows))
Just True
*Lies> isTrue (upd headsg (announce 0)) bKnows
Just False
*Lies> isTrue (upd headsg (announce 0)) (K b (Neg bKnows))
Just True
*Lies> isTrue (upd tailsg (announce 1)) bKnows
Just False
*Lies> isTrue (upd tailsg (announce 1)) (K b (Neg bKnows))
Just True
*Lies> isTrue (upd headsg (announce 1)) bKnows
Just False
*Lies> isTrue (upd headsg (announce 1)) (K b (Neg bKnows))
Just True
```

Note that since we did not use the manipulative update to model player a's announcement (although it is easy to implement in DEMO, of course) the resulting models are still S5-models.

```
Lies> isS5Model (upd headsg (announce 1))
True
Lies> isS5Model (upd headsg (announce 0))
True
Lies> isS5Model (upd tailsg (announce 1))
True
Lies> isS5Model (upd tailsg (announce 0))
True
```

This means that no actual misleading is taking place at all! This is actually very plausible because player b knows that player a's announcement might very well be false. This shows that lying only creates false belief if the person who lies is believed to be telling the truth.

Now we can use these action models to do a doxastic analysis of a game of Liar's Dice. The different possible games are:

1. Player a tosses tails and announces $\ddagger Tail$
2. Player a tosses heads and announces $\ddagger Tail$
3. Player a tosses tails and announces $\ddagger Head$ and player b passes
4. Player a tosses tails and announces $\ddagger Head$ and player b challenges
5. Player a tosses heads and announces $\ddagger Head$ and player b passes
6. Player a tosses heads and announces $\ddagger Head$ and player b challenges

The models for these games are:

```
game1, game2, game3, game4, game5, game6 :: EM
game1 = gsm (upd tailsg (announce 0))
game2 = gsm (upd headsg (announce 0))
game3 = gsm (upd (upd tailsg (announce 1)) pass)
game4 = gsm (upd (upd tailsg (announce 1)) challenge)
game5 = gsm (upd (upd headsg (announce 1)) pass)
game6 = gsm (upd (upd headsg (announce 1)) challenge)
```

We will now consider these six different cases in turn.

Game 1 is the game where player 1 tosses tails and admits this.

In this case both players stake one euro and player b wins the stakes, so in the end player a lost one euro and player b won one euro. This can be checked with DEMO:

```
*Lies> isTrue game1 (Eq (Reg c1) (ASum [I initCash1,I (-1)]))
Just True
*Lies> isTrue game1 (Eq (Reg c2) (ASum [I initCash2,I 1]))
Just True
```

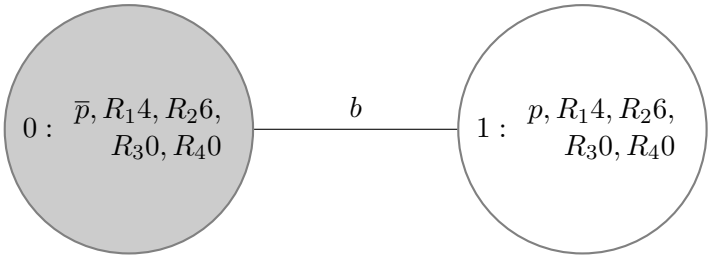
Player b doesn't get to know what the value of the coin was:

```
*Lies> isTrue game1 bKnows
Just False
```

The model for game 1 is:

```
*Lies> displayS5 game1
[0,1]
[p]
[R1,R2,R3,R4]
[(0,[]),(1,[p])]
[(0,[(R1,4),(R2,6),(R3,0),(R4,0)]),
 (1,[(R1,4),(R2,6),(R3,0),(R4,0)])]
(a,[[0],[1]])
(b,[[0,1]])
[0]
```

A picture of this model is below. There are two worlds, one where the toss was heads and one where it was tails. Player a can distinguish these worlds, player b cannot because player b never got to see the coin. In both worlds the cash of player a is 4 and that of player b is 6 euros, because the division of the stakes doesn't depend on the value of the coin. Reflexive arrows are not shown.



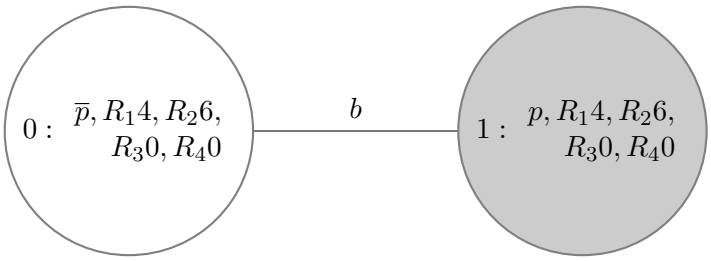
Game 2 is the game where player a falsely announces $\ddagger Head$. Just like in game 1, player a loses one euro and player b wins one euro, and player b doesn't get to know the value of the coin.

```
*Lies> isTrue game2 (Eq (Reg c1) (ASum [I initCash1,I (-1)]))
Just True
*Lies> isTrue game2 (Eq (Reg c2) (ASum [I initCash2,I 1]))
Just True
*Lies> isTrue game2 bKnows
Just False
```

The model for this game is almost the same as for game 1: the difference is that now the world where p is true is actual instead of the world where p is false.

```
*Lies> displayS5 game2
[0,1]
[p]
[R1,R2,R3,R4]
[(0,[]),(1,[p])]
[(0,[(R1,4),(R2,6),(R3,0),(R4,0)]),
 (1,[(R1,4),(R2,6),(R3,0),(R4,0)])]
(a,[[0],[1]])
(b,[[0,1]])
[1]
```

The picture of this model (reflexive arrows not shown) is:



The third game is the case where player a tosses tails but falsely announces $\ddagger Head$ and player b passes. In this case player a stakes two euros and player b stakes one euro, and player a gets to keep the stakes, so the final payoff is that player a wins one euro and player b loses one euro:

```
*Lies> isTrue game3 (Eq (Reg c1) (ASum [I initCash1,I 1]))
Just True
*Lies> isTrue game3 (Eq (Reg c1) (ASum [I initCash1,I 1]))
Just True
```

Player b passes, so the cup is never lifted and player b doesn't know the value of the coin:

```
*Lies> isTrue game3 bKnows
Just False
```

The model for this game is:

```
*Lies> displayS5 game3
[0,1]
[p]
[R1,R2,R3,R4]
[(0,[]),(1,[p])]
[(0,[(R1,6),(R2,4),(R3,0),(R4,0)]),
 (1,[(R1,6),(R2,4),(R3,0),(R4,0)])]
(a,[[0],[1]])
(b,[[0,1]])
[0]
```

This model has the same two worlds as the models for game 1 and 2 except for the changes in the player's cash.

In the fourth game, player a tosses tails but falsely announces $\ddagger Head$ and player b challenges player a. This means that both players stake one extra euro and then the cup is lifted and player b gets the stakes.

In this case player b does know the value of the coin:

```
*Lies> isTrue game4 bKnows
Just True
```

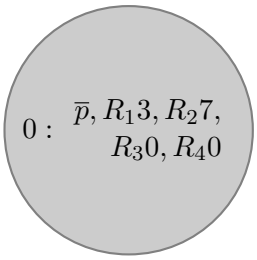
The payoffs are -2 euros for player a and 2 euros for player b:

```
*Lies> isTrue game4 (Eq (Reg c1) (ASum [I initCash1,I (-2)]))
Just True
*Lies> isTrue game4 (Eq (Reg c1) (ASum [I initCash1,I (-2)]))
Just True
```

The model for this game is:

```
*Lies> displayS5 game4
[0]
[p]
[R1,R2,R3,R4]
[(0,[])]
[(0,[(R1,3),(R2,7),(R3,0),(R4,0)])]
(a,[[0]])
(b,[[0]])
[0]
```

This model has only one world because none of the players consider any other world possible, because both players know the values of the coin. In this world p is false (because the toss was tails), player a's cash is 3 euros and player b's cash is 7 euros. A picture of this model is below.



The fifth game is the game where player a tosses heads and truthfully announces this and player b passes. In this case the cup isn't lifted so player b doesn't know the value of the coin again:

```
*Lies> isTrue game5 bKnows
Just False
```

The payoffs are 1 for player a and -1 for player b:

```
*Lies> isTrue game5 (Eq (Reg c1) (ASum [I initCash1,I 1]))
Just True
*Lies> isTrue game5 (Eq (Reg c2) (ASum [I initCash2,I (-1)]))
Just True
```

The model for game 5 has two worlds again because player b doesn't know the value of the coin.

```
*Lies> displayS5 game5
[0,1]
[p]
[R1,R2,R3,R4]
[(0,[]),(1,[p])]
[(0,[(R1,6),(R2,4),(R3,0),(R4,0)]),
 (1,[(R1,6),(R2,4),(R3,0),(R4,0)])]
(a,[[0],[1]])
(b,[[0,1]])
[1]
```

In game 6 player a tosses heads and truthfully announces this and player b challenges player a. In this case both players add one extra euro to the stakes, the cup is lifted and player a gets to keep the stakes. The model for this has one world where p is true, player a has 7 euros and player b has 3 euros.

```
*Lies> displayS5 game6
[0]
[p]
[R1,R2,R3,R4]
[(0,[p])]
[(0,[(R1,7),(R2,3),(R3,0),(R4,0)])]
(a,[[0]])
(b,[[0]])
[0]
```

In this case player b knows the value of the coin and the payoffs are 2 euros for player 1 and -2 euros for player 2:

```
*Lies> isTrue game6 bKnows
Just True
*Lies> isTrue game6 (Eq (Reg c1) (ASum [I initCash1,I 2]))
Just True
*Lies> isTrue game6 (Eq (Reg c2) (ASum [I initCash2,I (-2)]))
Just True
```

Bibliography

[BvEK06] J. van Benthem, J. van Eijck, and B. Kooi. Logics of communication and change. *Information and Computation*, 204(11):1620–1662, 2006.

[CH00] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. Of International Conference on Functional Programming (ICFP)*, ACM SIGPLAN, 2000.

[DvESW07] Hans van Ditmarsch, Jan van Eijck, Floor Sietsma, and Yanjing Wang. On the logic of lying. manuscript, CWI, 2007.

[Eijar] Jan van Eijck. Perception and change in update logic. In Jan van Eijck and Rineke Verbrugge, editors, *Games, Actions and Social Software*. Springer, 2011 (to appear).

[EWS10] Jan van Eijck, Yanjing Wang, and Floor Sietsma. Composing models. In Wiebe van der Hoek, editor, *Online Proceedings of LOFT 2010*, `http://loft2010.csc.liv.ac.uk/`, 2010.

[Ger99] J. Gerbrandy. *Bisimulations on Planet Kripke*. PhD thesis, ILLC, Amsterdam, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[J.E71] J.E.Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*. Academic Press, 1971.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland and P. Noordhoff, 1952.

[Pla89] J. A. Plaza. Logics of public communications. In M. L. Emrich, M. S. Pfeifer, M. Hadzikadic, and Z. W. Ras, editors, *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 201–216, 1989.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.