DEMO-S5

Jan van Eijck

March 11, 2014

Abstract

This paper documents a tiny program for epistemic model checking with S5 models in Haskell. The model update operations are public announcement and publicly observable factual change. The implementation is *much* more efficient than the earlier implementation documented in [vE07]. As examples, we implement the sum and product riddle, which is solved in a few seconds, and the parametrized muddy children problems, where the cases with up to ten children run in a matter of seconds.

Chapter 1

Models, Formulas, Announcements

The implementation is in literate programming style [Knu92], using Haskell [HT].

Declaration of the main module. If this module is loaded as object code (`ghci -fobject-code DEMO_S`) all computations are very fast.

```
module DEMO_S5 where

import Data.List
import EREL
```

The module `EREL` describes an implementation of relations and partitions [Eij14].

The basic concept is the logic of knowledge is that of epistemic uncertainty. If I am uncertain about whether a coin that has just been tossed is showing head or tail, this can be pictured as two situations related by my uncertainty. Such uncertainty relations are equivalences: If I am uncertain between situations s and t, and between situations t and r, this means I am also uncertain between s and r.

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

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

a,b,c,d,e :: Agent
a = Ag 0; b = Ag 1; c = Ag 2; d = Ag 3; e = 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 | S 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
  show (S 0) = "s"; show (S i) = "s" ++ show i
```

A datatype for epistemic models:

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

An example epistemic model:

```
example :: EpistM Int
example = Mo
 [0..3]
 [a,b,c]
 []
 [(a,[[0],[1],[2],[3]]),(b,[[0],[1],[2],[3]]),(c,[[0..3]])]
 [1]
```

In this model there are three agents and four possible worlds. The first two agents a and b can distinguish all worlds, and the third agent c confuses all of them. The actual world of the model is world 1.

Epistemic models can have more than one actual world. This is useful to cover several situations within a single model. Say, the situation where a coin has landed heads up and the situation where the same coin has landed tails up. Also, the result of a public announcement of something that is false in all actual worlds of a model is a model without actual worlds.

```
example2 :: EpistM Int
example2 = Mo
 [0..3]
 [a,b,c]
 [(0,[P 0,Q 0]),(1,[P 0]),(2,[Q 0]),(3,[])]
 [(a,[[0..3]]),(b,[[0..3]]),(c,[[0..3]])]
 [0..3]
```

Extracting an epistemic relation from a model:

```
rel :: Agent -> EpistM a -> Erel a
rel ag (Mo _ _ _ rels _) = table2fct rels ag

table2fct ::  Eq a => [(a,b)] -> a -> b
table2fct t = \ x -> maybe undefined id (lookup x t)
```

This gives:

Just a lookup function

```
*DEMO_S5> rel a example
[[0],[1],[2],[3]]
*DEMO_S5> rel c example
[[0,1,2,3]]
*DEMO_S5> rel d example
*** Exception: item not found
```

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 $|P| = k$, looks as follows:

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

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

Note that V is in fact a bijection, for $|\mathcal{P}(P)| = 2^k = |W|$. Each agent is completely ignorant, for all members of W are in the same partition block.

Generating Models for Blissful Ignorance

```
initM :: (Num state, Enum state) =>
         [Agent] -> [Prp] -> EpistM state
initM ags props = (Mo worlds ags val accs points)
  where
    worlds  = [0..(2^k-1)]
    k       = length props
    val     = zip worlds (sortL (powerList props))
    accs    = [ (ag,[worlds]) | ag  <- ags        ]
    points  = worlds
```

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)
```

This gives:

```
*DEMO_S5> initM [a,b,c,d] [P 0,Q 0]
Mo [0,1,2,3] [a,b,c,d] [(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
[(a,[[0,1,2,3]]),(b,[[0,1,2,3]]),
 (c,[[0,1,2,3]]),(d,[[0,1,2,3]])] [0,1,2,3]
```

Logical Forms The following datatype gives a logical form language for epistemic statements. Note that the type has a parameter for additional information. We will use that below for modelling epistemic riddles involving knowledge about numbers.

```
data Form a = Top
            | Info a
            | Prp Prp
            | Ng (Form a)
            | Conj [Form a]
            | Disj [Form a]
            | Kn Agent (Form a)
          deriving (Eq,Ord,Show)
```

A useful abbreviation:

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

Semantic interpretation for this logical form language:

```
isTrueAt :: Ord state =>
            EpistM state -> state -> Form state -> Bool
isTrueAt m w Top = True
isTrueAt m w (Info x) = w == x
isTrueAt
  m@(Mo worlds agents val acc points) w (Prp p) = let
   props = table2fct val w
  in
   elem p props
isTrueAt m w (Ng f) = not (isTrueAt m w f)
isTrueAt m w (Conj fs) = and (map (isTrueAt m w) fs)
isTrueAt m w (Disj fs) = or  (map (isTrueAt m w) fs)
isTrueAt
 m@(Mo worlds agents val acc points) w (Kn ag f) = let
    r = rel ag m
    b = bl r w
  in
    and (map (flip (isTrueAt m) f) b)
```

This treats the Boolean connectives as usual, and interprets knowledge as truth in all worlds in the current accessible equivalence block of an agent.

Truth in a model is defined as truth in all actual worlds of the model:

```
isTrue :: Ord a => EpistM a -> Form a -> Bool
isTrue m@(Mo worlds agents val acc points) f =
  all (\w -> isTrueAt m w f) points
```

The effect of a public announcement φ on an epistemic model is that the

set of worlds of that model gets limited to the worlds where φ is true, and the accessibility relations get restricted accordingly.

```
upd_pa :: Ord state =>
          EpistM state -> Form state -> EpistM state
upd_pa m@(Mo states agents val rels actual) f =
  (Mo states' agents val' rels' actual')
   where
   states' = [ s | s <- states, isTrueAt m s f ]
   val'    = [ (s, ps) | (s,ps) <- val, s `elem` states' ]
   rels'   = [(ag,restrict states' r) | (ag,r) <- rels ]
   actual' = [ s | s <- actual, s `elem` states' ]
```

A series of public announcement updates:

```
upds_pa ::  Ord state =>
            EpistM state -> [Form state] -> EpistM state
upds_pa = foldl upd_pa
```

Following [BvEK06], we represent changes in the world as substitutions. A substitution maps proposition letters to formulas.

Substitutions as functions, constructed from tables:

```
sub :: Eq a => [(Prp,Form a)] -> Prp -> Form a
sub subst p =
  if elem p (map (\ (x,_) -> x) subst) then
     table2fct subst p else (Prp p)
```

A public change update:

```
upd_pc :: Ord state => [Prp] -> EpistM state
                    -> [(Prp,Form state)] -> EpistM state
upd_pc ps m@(Mo states agents val rels actual) sb =
  (Mo states agents val' rels actual)
   where
   val' = [ (s, [p | p <- ps, isTrueAt m s (sub sb p)])
                                              | s <- states ]
```

A series of public change updates:

```
upds_pc :: Ord state => [Prp] -> EpistM state
                     -> [[(Prp,Form state)]] -> EpistM state
upds_pc ps = foldl (upd_pc ps)
```

Example substitution:

```
sexample =  [(P i, Kn a (Info i)) | i <- [0..3 ]]
             ++
            [(Q i, Kn b (Info i)) | i <- [0..3 ]]
             ++
            [(R i, Kn c (Info i)) | i <- [0..3 ]]
```

A list of propositions:

```
exampleprops = [P i | i <- [0..3 ]]
                ++
               [Q i | i <- [0..3 ]]
                ++
               [R i | i <- [0..3 ]]
```

This gives:

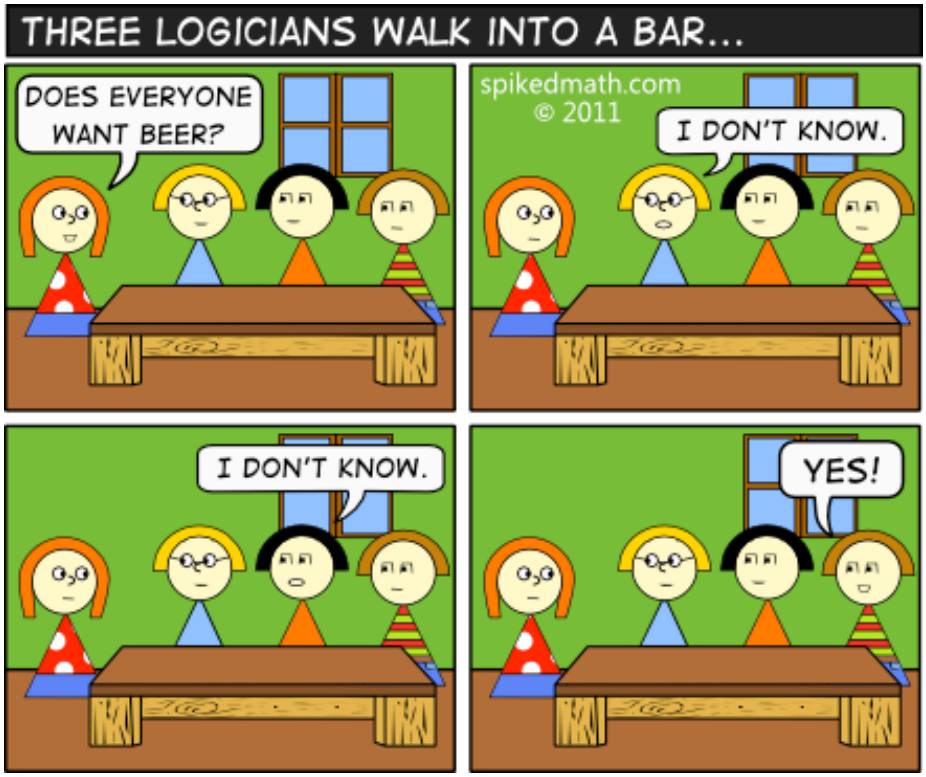
```
DEMO_S5> upd_pc exampleprops example sexample
Mo [0,1,2,3] [a,b,c] [(0,[p,q]),(1,[p1,q1]),(2,[p2,q2]),(3,[p3,q3])]
   [(a,[[0],[1],[2],[3]]),(b,[[0],[1],[2],[3]]),(c,[[0,1,2,3]])] [1]
*DEMO_S5>
```

Another way to represent factual change is as a change of world index. Call this public index change.

```
upd_pi :: (state -> state) -> EpistM state -> EpistM state
upd_pi f m@(Mo states agents val rels actual) =
  Mo
  (map f states)
  agents
  (map (\ (x,ps) -> (f x,ps)) val)
  (map (\ (ag,erel) -> (ag, map (map f) erel)) rels)
  (map f actual)
```

Chapter 2

Three Logicians



```
bools = [True,False]
```

Initialize the bar situation: they all know what they want but are ignorant about what the others want.

```
initBar :: EpistM (Bool,Bool,Bool)
initBar = Mo states [a,b,c] [] rels [(True,True,True)]
  where
  states = [ (b1,b2,b3) | b1 <- bools,
                          b2 <- bools,
                          b3 <- bools ]
  rela = (a,[[(True,x,y)  | x <- bools, y <- bools],
             [(False,x,y) | x <- bools, y <- bools]])
  relb = (b,[[(x,True,y)  | x <- bools, y <- bools],
             [(x,False,y) | x <- bools, y <- bools]])
  relc = (c,[[(x,y,True)  | x <- bools, y <- bools],
             [(x,y,False) | x <- bools, y <- bools]])
  rels = [rela,relb,relc]
```

Statements of ignorance and knowledge:

```
allBeer :: Form (Bool,Bool,Bool)
allBeer = Info (True,True,True)

ignA, ignB, ignC :: Form  (Bool,Bool,Bool)
ignA = Conj [Ng (Kn a allBeer), Ng (Kn a (Ng allBeer))]
ignB = Conj [Ng (Kn b allBeer), Ng (Kn b (Ng allBeer))]
ignC = Conj [Ng (Kn c allBeer), Ng (Kn c (Ng allBeer))]

knowC, knowC' :: Form  (Bool,Bool,Bool)
knowC  = Kn c allBeer
knowC' = Kn c (Ng allBeer)
```

Result of first update:

```
barModel1 = upd_pa initBar ignA
```

This gives:

```
*DEMO_S5> barModel1
Mo [(True,True,True),(True,True,False),(True,False,True),(True,False,False)]
[a,b,c] []
[(a,[[(True,True,True),(True,True,False),(True,False,True),(True,False,False)]]),
 (b,[[(True,True,True),(True,True,False)],[(True,False,True),(True,False,False)]]),
 (c,[[(True,True,True),(True,False,True)],[(True,True,False),(True,False,False)]])]
[(True,True,True)]
```

Result of second update:

```
barModel2 = upd_pa barModel1 ignB
```

This gives:

```
*DEMO_S5> barModel2
Mo [(True,True,True),(True,True,False)]
[a,b,c] []
[(a,[[(True,True,True),(True,True,False)]]),
 (b,[[(True,True,True),(True,True,False)]]),
 (c,[[(True,True,True)],[(True,True,False)]])]
[(True,True,True)]
```

Result of third update:

```
barModel3 = upd_pa barModel2 knowC
```

This gives:

```
*DEMO_S5> barModel3
Mo [(True,True,True)] [a,b,c] []
[(a,[[(True,True,True)]]),
 (b,[[(True,True,True)]]),
 (c,[[(True,True,True)]])]
[(True,True,True)]
```

Or the third logician could have said 'no', for 'I know that not all of us want beer'.

```
barModel3' = upd_pa barModel2 knowC'
```

This gives:

```
*DEMO_S5> barModel3'
Mo [(True,True,False)] [a,b,c] []
[(a,[[(True,True,False)]]),
 (b,[[(True,True,False)]]),
 (c,[[(True,True,False)]])] []
```

Chapter 3

Sum and Product

We illustrate the working of the update mechanism on a famous epistemic puzzle. The following Sum and Product riddle was stated by the Dutch mathematican Hans Freudenthal in a Dutch mathematics journal in 1969.

> A says to S and P: I have chosen two integers x, y such that $1 < x < y$ and $x + y \leq 100$. In a moment, I will inform S only of $s = x + y$, and P only of $p = xy$. These announcements remain private. You are required to determine the pair (x, y). He acts as said. The following conversation now takes place:
>
> 1. P says: "I do not know the pair."
> 2. S says: "I knew you didn't."
> 3. P says: "I now know it."
> 4. S says: "I now also know it."
>
> Determine the pair (x, y).

This was solved by combinatorial means in a later issue of the journal.

There is also a version by John McCarthy (see `http://www-formal.stanford.edu/jmc/puzzles.htm`).

A model checking solution with DEMO [vE07] (based on a DEMO program written by Ji Ruan) was presented in [DRV05]. The present program is an optimized version of that solution.

The list of candidate pairs:

```
pairs :: [(Int,Int)]
pairs  = [ (x,y) |  x <- [2..100], y <- [2..100],
                    x < y, x+y <= 100 ]
```

The initial epistemic model is such that a (representing S) cannot distinguish number pairs with the same sum, and b (representing P) cannot distinguish number pairs with the same product. Instead of using a valuation, we use number pairs as worlds.

```
msnp :: EpistM (Int,Int)
msnp = (Mo pairs [a,b] [] acc pairs)
  where
  acc   = [ (a, [ [ (x1,y1) | (x1,y1) <- pairs,
                               x1+y1 == x2+y2 ] |
                                 (x2,y2) <- pairs ] ) ]
          ++
          [ (b, [ [ (x1,y1) | (x1,y1) <- pairs,
                               x1*y1 == x2*y2 ] |
                                 (x2,y2) <- pairs ] ) ]
```

The statement by b that he does not know the pair:

```
statement_1 =
  Conj [ Ng (Kn b (Info p)) | p <- pairs ]
```

To check this statement is expensive. A computationally cheaper equivalent statement is the following (see [DRV05]).

```
statement_1e =
  Conj [ Info p `impl` Ng (Kn b (Info p)) | p <- pairs ]
```

In Freudenthal's story, the first public announcement is the statement where b confesses his ignorance, and the second public announcement is the statement by a about her knowledge about b's state of knowledge *before* that confession. We can wrap the two together in a single statement to the effect that initially, a knows that b does not know the pair. This gives:

```
k_a_statement_1e = Kn a statement_1e
```

The second announcement proclaims the statement by b that now he knows:

```
statement_2 =
  Disj [ Kn b (Info p) | p <- pairs ]
```

Equivalently, but computationally more efficient:

```
statement_2e =
  Conj [ Info p `impl` Kn b (Info p) | p <- pairs ]
```

The final announcement concerns the statement by a that now she knows as well.

```
statement_3 =
  Disj [ Kn a (Info p) | p <- pairs ]
```

In the computationally optimized version:

```
statement_3e =
  Conj [ Info p `impl` Kn a (Info p) | p <- pairs ]
```

The solution:

```
solution = upds_pa msnp
           [k_a_statement_1e,statement_2e,statement_3e]
```

This is checked in a matter of seconds:

```
*DEMO_S5> solution
Mo [(4,13)] [a,b] [(a,[[(4,13)]]),(b,[[(4,13)]])] [(4,13)]
```

Chapter 4

Muddy Children

Now let us look at how efficient this implementation is for model checking of muddy children situations.

Let's start with three children, two muddy:

```
init_mud :: EpistM (Bool,Bool,Bool)
init_mud = Mo states [a,b,c] [] rels [(False,True,True)]
  where
  states = [ (m1,m2,m3) | m1 <- bools,
                          m2 <- bools,
                          m3 <- bools ]
  rela = (a,[[(True,x,y),(False,x,y)] | x <- bools,
                                        y <- bools])
  relb = (b,[[(x,True,y),(x,False,y)] | x <- bools,
                                        y <- bools])
  relc = (c,[[(x,y,True),(x,y,False)] | x <- bools,
                                        y <- bools])
  rels = [rela,relb,relc]
```

Statement of father:

```
father :: Form (Bool,Bool,Bool)
father = Ng (Info (False,False,False))
```

a knows:

```
kn_a = Disj [Kn a (Disj [Info (True,x,y) | x <- bools,
                                           y <- bools]),
             Kn a (Disj [Info (False,x,y) | x <- bools,
                                            y <- bools ])]
```

b knows:

```
kn_b = Disj [Kn b (Disj [Info (x,True,y) | x <- bools,
                                           y <- bools]),
             Kn b (Disj [Info (x, False,y) | x <- bools,
                                             y <- bools ])]
```

c knows:

```
kn_c = Disj [Kn c (Disj [Info (x,y,True) | x <- bools,
                                           y <- bools]),
             Kn c (Disj [Info (x,y,False) | x <- bools,
                                            y <- bools ])]
```

They all don't know:

```
dont_know :: Form (Bool,Bool,Bool)
dont_know = Conj [Ng kn_a, Ng kn_b, Ng kn_c]
```

Three consecutive updates of the initial model:

```
mod1 = upd_pa init_mud father
mod2 = upd_pa mod1 dont_know
mod3 = upd_pa mod2 (Conj [kn_b,kn_c])
```

The solution:

```
solution3 = upds_pa init_mud
          [father,dont_know,Conj [kn_b,kn_c]]
```

Now let's generalize this, by making the number of children a parameter. We look at examples with n children, with $n-1$ muddy.

We build the models over lists of booleans as states. This is to be read as follows: if there are n elements in the list, then there are n children. If item i in the list equals `True` then the i-th child is muddy, if this item equals `False` then this child is not muddy.

Here is a function for creating all possible length n lists of booleans:

```
bTables :: Int -> [[Bool]]
bTables 0 = [[]]
bTables n = map (True:) (bTables (n-1))
            ++ map (False:) (bTables (n-1))
```

Initializing a model for n children is done by taking as states all possible Boolean lists of length n, and setting the relations as appropriate: child i cannot distinguish two states that only differ in their i-th position. Note that there is no need for valuation information, as the muddiness info is encoded in the state lists. It is assumed that the first child is not muddy, but all other children are.

```
initN :: Int -> EpistM [Bool]
initN n = Mo states agents [] rels points where
  states = bTables n
  agents = map Ag [1..n]
  rels = [(Ag i, [[tab1++[True]++tab2,tab1++[False]++tab2] |
                  tab1 <- bTables (i-1),
                  tab2 <- bTables (n-i) ]) | i <- [1..n] ]
  points = [False: take (n-1) (repeat True)]
```

The statement of the father, parametrized for the number of children:

```
fatherN :: Int -> Form [Bool]
fatherN n = Ng (Info (take n (repeat False)))
```

Player i knows:

```
kn :: Int -> Int -> Form [Bool]
kn n i = Disj [Kn (Ag i) (Disj [Info (tab1++[True]++tab2) |
                        tab1 <- bTables (i-1),
                        tab2 <- bTables (n-i) ]),
               Kn (Ag i) (Disj [Info (tab1++[False]++tab2) |
                        tab1 <- bTables (i-1),
                        tab2 <- bTables (n-i) ])]
```

The children do not know their state:

```
dont :: Int -> Form [Bool]
dont n = Conj [Ng (kn n i) | i <- [1..n] ]
```

General knowledge of all children except the first:

```
knowN n = Conj [kn n i | i <- [2..n] ]
```

The parametrized solution:

```
solveN :: Int -> EpistM [Bool]
solveN n = upds_pa (initN n) (f:istatements ++ [knowN n])
  where
  f = fatherN n
  istatements = take (n-2) (repeat (dont n))
```

Now try this out for $2, 3, 4, 5 \ldots$ children.

Chapter 5

N Prisoners and a Lightbulb

100 (or, let us say, n) prisoners are together in the prison dining area. They are told that they will be all put in isolation cells and then will be interrogated one by one in a room containing a light with an on/off switch. The prisoners may communicate with one another by toggling the light-switch (and that is the only way in which they can communicate). The light is initially switched off. There is no fixed order of interrogation, or fixed interval between interrogations, and the same prisoner may be interrogated again at any stage. When interrogated, a prisoner can either do nothing, or toggle the light-switch, or announce that all prisoners have been interrogated. If that announcement is true, the prisoners will (all) be set free, but if it is false, they will all be executed. While still in the dining room, and before the prisoners go to their isolation cells, can the prisoners agree on a protocol that will set them free (assuming that at any stage every prisoner will be interrogated again sometime)?

Here is a possible protocol: The n prisoners appoint one amongst them as the counter. All non-counting prisoners act according to the following protocol: the first time they enter the room when the light is off, they turn it on; on all other occasions, they do nothing. The counter follows a different protocol. The first $n-2$ times that the light is on when he enters the interrogation room, he turns it off. The next time he enters the room when the light is on, he (truthfully) announces that everybody has been interrogated.

To model this, we will use models over `(Int,[Int])`, where the first element gives the number of prisoner that have been counted, and the second element gives their list.

In the initial model no prisoners have been counted yet. We only represent the knowledge of the prisoner who does the counting (prisoner 0).

```
initPrison :: EpistM (Int,[Int])
initPrison = let x = (0,[]) in
  Mo
   [x]
   [a]
   [(x,[])]
   [(a,[[x]])]
   [x]
```

Mark prisoner i as counted:

```
counted :: Int -> [Int] -> [Int]
counted i xs = if elem i xs then xs else insert i xs
```

Encoding for "the light is on:"

```
light = [P 0]
```

Interrogation of the counter (= prisoner 0). If the light is on, then switch it off and count one extra prisoner, otherwise do nothing.

```
interrog :: EpistM (Int,[Int]) -> Int -> EpistM (Int,[Int])
interrog (Mo [(i,is)] [a] val _ _) 0 = let
  ys = apply val (i,is)
  i' = if ys == light then i+1 else i
  x  = (i',is)
 in
  Mo [x] [a] [(x,[])] [(a,[[x]])] [x]
```

Interrogation of prisoner $i \neq 0$: if the light is on or i has already been counted then i does nothing, otherwise i switches on the light. We assume prisoner i remembers whether (s)he has switched on the light before.

```
interrog (Mo [(i,is)] [a] val _ _) k = let
  ys  = apply val (i,is)
  is' = if ys == light then is else (counted k is)
  ys' = if elem k is then ys else light
  x   = (i,is')
 in
  Mo [x] [a] [(x,ys')] [(a,[[x]])] [x]
```

Interrogation of a list of prisoners:

```
inter :: EpistM (Int,[Int]) -> [Int] -> EpistM (Int,[Int])
inter = foldl' interrog
```

Check whether k prisoners have been counted: check the index after adding 1 (for prisoner 0, who does the counting).

```
check k (Mo [(i,_)] _ _ _ _) = k == i+1
```

Representation of interrogation sequeces as infinite streams of Ints. Here is an example, of continued interrogation of k prisoners, always in the same order.

```
process :: Int -> [Int]
process k = cycle [0..k-1]
```

Representation of the counting protocol for k prisoners and an infinite interrogation sequence.

```
protocol :: Int -> [Int] -> [EpistM (Int,[Int])]
protocol k = protocol' initPrison where
  protocol' m (i:is)
    | check k m = [m]
    | otherwise =  m : protocol' (interrog m i) is
```

A measure for calculating how long it takes before the counter (prisoner 0) knows that all have been interrogated.

```
measure :: Int -> [Int] -> Int
measure k pr = length (protocol k pr) - 1
```

If one prisoner per day gets interrogated, and they get interrogated in the same order starting from 0 always, then the protocol for 100 prisoners takes over 27 years before it terminates (discarding leap years):

```
*DEMO_S5> quotRem (measure 100 (cycle [0..99])) 365
(27,46)
```

For further information, see [DvEW10a, DvEW10b].

Chapter 6

Update Models, Generic S5 Update

Update models for S5 are like epistemic S5 models, but with the valuation replaced by a precondition function that assigns formulas to worlds.

```
data UpdateM state = UM
  [state]
  [Agent]
  [(state,Form state)]
  [(Agent,Erel state)]
  [state] deriving (Eq,Show)
```

The list of agents in an update model should match that of the input epistemic model. This can be achieved by means of the following type:

```
type FUM state = [Agent] -> UpdateM state
```

An example `FUM`:

```
fum1 :: FUM Int
fum1 = \ ags -> UM
  [0,1]
  ags
  [(0,Disj [Prp (P 0),Prp (Q 0)]),(1,Top)]
  ((a,[[0],[1]]): [(b,[[0,1]]) | b <- ags \\ [a] ])
  [0]
```

In this example, a learns that $p \vee q$, while all other agents remain unaware of this.

Updating with an action model:

```
upd :: Ord state =>
       EpistM state -> FUM state -> EpistM (state,state)
upd m@(Mo states agents val rels actual) fum =
  Mo states' agents' val' rels' actual'
  where
  UM events agents' pre susp aevent = fum agents
  states' = [ (s,e) | s <- states, e <- events,
                      isTrueAt m s (table2fct pre e) ]
  val'    = [ ((s,e),props) | (s,props) <- val,
                              e        <- events,
                              elem (s,e) states'  ]
  rels'   =  [ (ag1, restrictedProd r s states') |
                                 (ag1,r) <- rels,
                                 (ag2,s) <- susp,
                                  ag1 == ag2      ]
  actual' = [ (a,e) | a <- actual,
                      e <- aevent, elem (a,e) states' ]
```

This gives:

```
*DEMO_S5> upd example2 fum1
Mo [(0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,1)] [a,b,c]
[((0,0),[p,q]),((0,1),[p,q]),((1,0),[p]),((1,1),[p]),
```

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

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

```
convrt :: Eq state =>
          EpistM state -> EpistM Int
convrt (Mo states agents val rels actual) =
 Mo states' agents val' rels' actual'
   where
    states' = map f states
    val'    = map (\ (x,y) -> (f x,y)) val
    rels'   = map (\ (x,r) -> (x, map (map f) r)) rels
    actual' = map f actual
    f       = table2fct (zip states [0..])
```

This gives:

```
*DEMO_S5> convrt $ upd example2 fum1
Mo [0,1,2,3,4,5,6] [a,b,c]
[(0,[p,q]),(1,[p,q]),(2,[p]),(3,[p]),(4,[q]),(5,[q]),(6,[])]
[(a,[[0,2,4],[1,3,5,6]]),
 (b,[[0,1,2,3,4,5,6]]),
 (c,[[0,1,2,3,4,5,6]])]
[0,2,4]
```

Bibliography

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

[DRV05] Hans van Ditmarsch, Ji Ruan, and Rineke Verbrugge. Model checking sum and product. In Shichao Zhang and Ray Jarvis, editors, *AI 2005: Advances in Artificial Intelligence: 18th Australian Joint Conference on Artificial Intelligence*, volume 3809 of *Lecture Notes in Computer Science*, pages 790–795. Springer-Verlag GmbH, 2005.

[DvEW10a] Hans van Ditmarsch, Jan van Eijck, and William Wu. One hundred prisoners and a lightbulb — logic and computation. In F. Lin and U. Sattler, editors, *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 90 – 100, Toronto, Canada, May 2010.

[DvEW10b] Hans van Ditmarsch, Jan van Eijck, and William Wu. Verifying one hundred prisoners and a lightbulb. *Journal of Applied Non-classical Logics*, pages 173–191, 2010.

[Eij14] Jan van Eijck. Relations, equivalences, partitions. Technical report, CWI, Amsterdam, 2014. Available from `http://www.cwi.nl/~jve/software/demo_s5`.

[HT] The Haskell Team. The Haskell homepage. `http://www.haskell.org`.

[Knu92] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[vE07] Jan van Eijck. DEMO — a demo of epistemic modelling. In Johan van Benthem, Dov Gabbay, and Benedikt Löwe, editors, *Interactive Logic — Proceedings of the 7th Augustus de Morgan Workshop*, number 1 in Texts in Logic and Games, pages 305–363. Amsterdam University Press, 2007.