Representing Public Announcement Updates

Jan van Eijck

jve@cwi.nl

August 5, 2008

Abstract

Public announcement is a means to create common knowledge: if φ is publicly announced to a set of agents, then every agent knows φ, every agent knows that every agent knows that φ, and so on.
We will first look at definitions, and then turn to implementation.

Public Announcement as an Update

Public announcements $[\varphi]$ change knowledge states, so their semantics can be given as a function from Kripke models to Kripke models:

$$M \mapsto M \mid \varphi$$

$M \mid \varphi$ given by:

$$\text{if } M = (W, V, R) \text{ then } M \mid \varphi = (W', V', R')$$

with

$$\begin{aligned} W' &= \{w \in W \mid M, w \models \varphi\} \\ V' &= V \upharpoonright W' \\ R' &= \{w \xrightarrow{a} w' \mid w \xrightarrow{a} w' \in R, w, w' \in W'\} \end{aligned}$$

Effect on Actual Worlds

A pointed Kripke model is a quadruple $M = (W, V, R, U)$ with (W, V, R) a Kripke model, and $U \subseteq W$ a set of points.

Intention: the actual world is among U.

Extension of the definition $M \mid \varphi$ to pointed models:

$$(W, V, R, U) \mid \varphi = (W', V', R', U')$$

where (W', V', R') is as above, and

$$U' = \{u \in U \mid (W, V, R), u \models \varphi\}$$

Public Announcement with Falsehood

φ is a falsehood in pointed model $M = (W, V, R, U)$ if

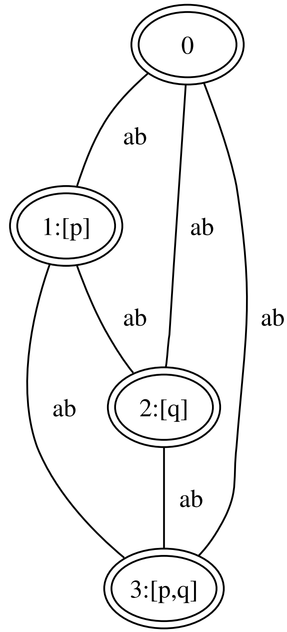
$$(W, V, R), u \not\models \varphi$$

for all $u \in U$.

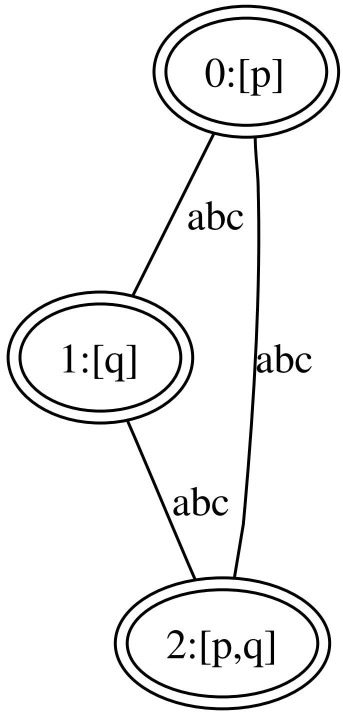
The result of updating with a falsehood is an inconsistent pointed model, i.e., a pointed model of the form $(W', V', R', \emptyset)$.

Learning that $p \vee q$ by public announcement

Initial model: a and b ignorant about p and q, and no possibility as yet ruled out:



Result of public announcement that $p \vee q$:



Module Declaration

```
module RPAU

where
import List
import HFKR
```

First your homework . . .

```
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
```

Example Epistemic Model

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

Extracting domain, relations, and valuation from an epistemic model

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

rel :: Agent -> EpistM a -> Rel a
rel a (Mo states agents val rels actual) =
      [ (x,y) | (agent,x,y) <- rels, a == agent ]

valuation ::  EpistM a -> [(a,[Prop])]
valuation (Mo _ _ val  _ _ ) = val
```

```
RPAU> rel a s5example
[(0,0),(1,1),(2,2),(3,3)]
RPAU> rel b s5example
[(0,0),(1,1),(2,2),(3,3)]
RPAU> rel c s5example
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),
 (2,0),(2,1),(2,2),(2,3),(3,0),(3,1),(3,2),(3,3)]
RPAU>  isS5 (dom s5example) (rel a s5example)
True
```

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 $[a]_R = \{b \in A \mid (a, b) \in R\}$.

Implementation:

```
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 val rels actual) =
  show states :
  show val    :
  map show [ (a, (rel2partition states) (rel a m))
                                     | a <- agents ]
  ++
  [show actual]
```

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

Example Display

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

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

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.

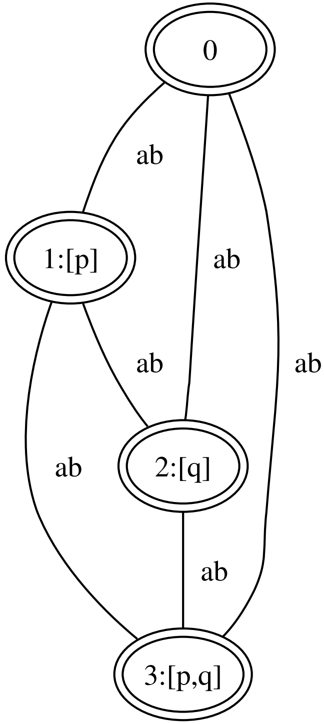
A Kripke model where every agent from agent set A is in blissful ignorance about a (finite) set of propositions P, with $|P| = k$, looks as follows:

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

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

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

Blissful Ignorance – Example



Generating Models for Blissful Ignorance

```
initM :: [Agent] -> [Prop] -> EpistM Integer
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,st1,st2) | ag  <- ags,
                              st1 <- worlds,
                              st2 <- worlds      ]
    points = worlds
```

`powerList`, `sortL`, `zip`: see below.

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)
```

zip

`zip` is a predefined function for zipping two lists together. Home-made version:

```
zip :: [a] -> [b] -> [(a,b)]
zip xs [] = []
zip [] ys = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

This gives:

```
RPAU> zip [0..2^3-1] (sortL (powerList [P 1,P 2,P 3]))
[(0,[]),(1,[p1]),(2,[p2]),(3,[p3]),(4,[p1,p2]),
 (5,[p1,p3]),(6,[p2,p3]),(7,[p1,p2,p3])]
```

General Knowledge

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

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

Implementation:

```
genK :: Ord state => [(Agent,state,state)]
                      -> [Agent] -> Rel state
genK r ags = [ (x,y) | (a,x,y) <- r, a `elem` 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 most important closures of relations:

- the reflexive closure,
- the symmetric closure,
- the transitive closure,
- 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.

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

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

```
RPAU> rtc [1,2,3] [(1,2),(2,3)]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

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)
```

Representing Formulas

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

CK is the operator for common knowledge.

Example formulas

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

Note the following type difference:

```
RPAU> :t (P 0)
P 0 :: Prop
RPAU> :t p
p :: Form
```

```haskell
instance Show Form where
  show Top            = "T"
  show (Prop 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
```

This gives:

```
RPAU> CK [a..c] (Disj[p,K a (Neg p)])
C[a,b,c]v[p,[a]-p]
```

Evaluation

```
isTrueAt :: Ord state =>
     EpistM state -> state -> Form -> Bool
```

Your homework for today.

Evaluating the State of Bliss

```
test1 = isTrueAt
  (initM [a..c] [P 0]) 0
  (CK [a..c] (Neg (K a p)))
```

Truth in a Model

Use the function isTrueAt to implement a function that checks for truth at all the designated states of an epistemic model:

```
isTrue :: Ord state => EpistM state -> Form -> Bool
isTrue m@(Mo worlds agents val acc points) f =
  and [ isTrueAt m s f | s <- points ]
```

Another test of initM

```
test2 = isTrue
        (initM [a..c] [P 0])
        (CK [a..c] (Neg (K a p)))
```

Finally: Public Announcement Update

```
upd_pa :: Ord state =>
          EpistM state -> Form -> 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,p) | (s,p) <- val,
                      s `elem` states'   ]
   rels'   = [(a,x,y) | (a,x,y) <- rels,
                        x `elem` states',
                        y `elem` states' ]
   actual' = [ s | s <- actual, s `elem` states' ]
```

Examples

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

```
RPAU> displayS5 m0
[0,1,2,3]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
[0,1,2,3]
```

```
RPAU> displayS5 (upd_pa m0 (Disj [p,q]))
[1,2,3]
[(1,[p]),(2,[q]),(3,[p,q])]
(a,[[1,2,3]])
(b,[[1,2,3]])
(c,[[1,2,3]])
[1,2,3]
```

Generated Submodels

```
gsm :: Ord state => EpistM state ->  EpistM state
gsm (Mo states ags val rel points) =
  (Mo states' ags val' rel' points)
   where
   states' = closure rel ags points
   val'    = [(s,props) | (s,props)     <- val,
                          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 $\{t \mid s \xrightarrow{b} t \in R, s \in S, b \in B\}$. Implementation:

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

Homework for today

Implement the function `isTrueAt` for checking the truth of a formula in a state in an epistemic model.

You should use induction on the structure of the formula, of course.

Next page gives the skeleton of the definition.

```
isTrueAt :: Ord state =>
     EpistM state -> state -> Form -> Bool
isTrueAt m w Top = ...
isTrueAt
 m@(Mo worlds agents val acc points) w (Prop p) = ...
isTrueAt m w (Neg f)   = ...
isTrueAt m w (Conj fs) = ...
isTrueAt m w (Disj fs) = ...
isTrueAt
 m@(Mo worlds agents val acc points) w (K ag f) = ...
isTrueAt
 m@(Mo worlds agents val acc points) w (CK ags f) = ...
```

Tomorrow

- Bisimulations
- Computing bisimulation-minimal models
- Action models
- Updating with an action model