Functional Programming
Epistemic Model Checking

May Lee
Dimitrios Koutsoulis

June 30, 2018

1 Introduction

We implemented the epistemic model checking for Kripke models and succint Kripke models (mental models). We also implemented translating functions between the two. Note that the translation from Kripke to mental models introduces nominal propositions whose sole purpose is to identify a world each. Given that we represent agents, worlds and propositions of Kripke models as positive integers, we let the nominal propositions be the negations of the worlds they represent. As such translating back and forth from a Kripke model nets us one whose universe of propositions is a strict superset of the original. Still if we restrict it to the original universe, it is isomorphic to the original Kripke model.

2 Code

2.1 Propositional Logic

Show instance for propositional formulas (includes the knowledge operator).

```
type Proposition = Integer
data Form = P Proposition | Neg Form | Conj Form Form | Disj Form Form | Top | Bottom | Knows Agent Form deriving (Eq,Ord)
instance Show Form where
  showsPrec _ (P name) = showString $ show name
  showsPrec p (Neg formula) = showParen (p > 3) $
    showString "not " ∘ showsPrec 3 formula
  showsPrec p (Conj lhs rhs) = showParen (p > 2) $
    showsPrec 3 lhs ∘ showString " and " ∘ showsPrec 2 rhs
  showsPrec p (Disj lhs rhs) = showParen (p > 1) $
    showsPrec 2 lhs ∘ showString " or " ∘ showsPrec 1 rhs
  showsPrec p Top = showString "T"
  showsPrec p Bottom = showString "B"
  showsPrec p (Knows agent formula) = showParen (p > 3) $
    showString ("K" ++ (show agent) ++ " " ) ∘ showsPrec 3 formula
```

The definition of the propositional logic satisfaction function adapted from the slides.

```
satisfies :: Assignment → Form → Bool
satisfies v (P k)      = k `elem` v
satisfies v (Neg f)    = not (satisfies v f)
satisfies v (Conj f g) = satisfies v f && satisfies v g
satisfies v (Disj f g) = satisfies v f || satisfies v g
satisfies _ Top        = True
```

```
satisfies _ Bottom     = False
```

Function "make_beta" that given AP_M, AP_W and the valuations of all worlds, computes β_M which we use to discard valuations that do not correspond to any possible world.

```
make_beta :: [Proposition] → [Proposition] → [(World, Valuation)] → Form
make_beta ap_M ap_W valuations = make_beta2 ap_M ap_W ap_W valuations

make_beta2 :: [Proposition] → [Proposition] → [Proposition] → [(World, Valuation)] → Form
make_beta2 _ [] ap_W _ = uExists ap_W
make_beta2 ap_M (prop_w:restProps) ap_W (val:valuations) = Conj (Disj (Neg (P prop_w)) (desc ap_M (snd val))) (make_beta2 ap_M restProps ap_W valuations)
```

- uExists constructs the formula that corresponds to the exclusive existential !∃. ap is the list of all propositions.
- possibleWorld is the conjunction of implications from nominal propositions to the descriptions of the world that they denote. So that if some p_w is true for an interpretation I, then if I is to satisfy the conjunct, it must also satisfy $V(w)$

```
uExists :: [Proposition] → Form
uExists ap = uExists2 [(x, remove x ap) | x ← ap]

uExists2 :: [(Proposition, [Proposition])] → Form
uExists2 [] = Bottom
uExists2 ((active_prop, inactive_props):rest) = Disj (possibleWorld active_prop inactive_props) (uExists2 rest)

possibleWorld :: Proposition → [Proposition] → Form
possibleWorld active_prop [] = P active_prop
possibleWorld active_prop (inactive_prop:rest) = Conj (Neg (P inactive_prop)) (possibleWorld active_prop rest)
```

The description of a valuation V is the conjunction of the propositions true in V along with the negations of those not true in V.

```
desc :: [Proposition] → [Proposition] → Form
desc ap props = desc2 ((nub ap) \\ props) props

desc2 :: [Proposition] → [Proposition] → Form
desc2 out_props (in_prop:in_props) = Conj (P in_prop) (desc2 out_props in_props)
desc2 (out_prop:out_props) []      = Conj (Neg (P out_prop)) (desc2 out_props [])
desc2 [] []                        = Top
```

2.2 Programs

Definition of the program. Note that the Universal Program can reach any world from any world while the

```
data Program = AssignTo Proposition Form
             | Question Form
             | Semicolon Program Program
             | Cap Program Program
             | Cup Program Program
             | IdleProgram
             | UniversalProgram
            deriving (Eq,Show)

remove element list = filter (λe → e/=element) list
```

```
runProgram :: Form → [Proposition] → Program → Assignment → Assignment → Bool
runProgram beta _ (AssignTo prop formula) a_1 a_2        | satisfies a_1 formula = (sort $ nub $ prop:a_1) == (sort $ nub a_2)
                                                         | otherwise             = (sort $ nub (remove prop a_1)) == (sort $ nub a_2)
runProgram beta _ (Question formula) a_1 a_2         = ((sort $ nub a_1) == (sort $ nub a_2)) && satisfies a_1 formula
runProgram beta ap (Semicolon pi_1 pi_2) a_1 a_2     =
          not (all (λx → not ((runProgram beta ap pi_1 a_1 x) && (runProgram beta ap pi_2 x a_2))) [ assignment | assignment ← (allAssignmentsFor ap), satisfies assignment beta] )
runProgram beta ap (Cap pi_1 pi_2) a_1 a_2           = (runProgram beta ap pi_1 a_1 a_2) && (runProgram beta ap pi_2 a_1 a_2)
runProgram beta ap (Cup pi_1 pi_2) a_1 a_2           = (runProgram beta ap pi_1 a_1 a_2) || (runProgram beta ap pi_2 a_1 a_2)
runProgram beta _ IdleProgram _ _                    = False
runProgram beta _ UniversalProgram _ _               = True

programSet :: [Proposition] → Program
programSet [] = IdleProgram
programSet (current_prop:rest) =
   Semicolon (Cup (AssignTo current_prop Top) (AssignTo current_prop Bottom)) (programSet rest)
```

2.3 Mental Model

Here follows the definition of the mental model and its model checking algorithm.

```
data SuccEpistM = Mo
       [Proposition]
       Form
       [(Agent, Program)]
       deriving (Eq,Show)

modelCheck :: SuccEpistM → Assignment → Form → Bool
modelCheck model world (P k)            = k `elem` world
modelCheck model world (Neg f)          = not (modelCheck model world f)
modelCheck model world (Conj f g)       = (modelCheck model world f) && (modelCheck model world g)
modelCheck model world (Disj f g)       = (modelCheck model world f) || (modelCheck model world g)
modelCheck (Mo ap beta programs) world (Knows agent f)
                                        = case (lookup agent programs) of
                                             Just program_a → all (λx → (not (runProgram beta ap program_a world  x))
                                                      || (modelCheck (Mo ap beta programs) x f) ) [ assignment | assignment ← (allAssignmentsFor ap), satisfies assignment beta]
                                             Nothing → False
```

2.4 Kripke Model

Definitions for the Kripke alongside its model checking algorithm.

```
type World = Integer
type Valuation = [Proposition]
type Relation = [(World, [World])]

data KripkeM = KMo
       [(Agent, Relation)]
       [(World, Valuation)]
       deriving (Eq,Show)
```

```
modelCheckKrpk :: KripkeM → World → Form → Bool
modelCheckKrpk (KMo relations valuations) world (P k) = case (lookup world valuations) of
                                          Just valuation → k `elem` valuation
                                          Nothing → False
modelCheckKrpk model world (Neg f)          = not (modelCheckKrpk model world f)
modelCheckKrpk model world (Conj f g)       = (modelCheckKrpk model world f) && (modelCheckKrpk model world g)
modelCheckKrpk model world (Disj f g)       = (modelCheckKrpk model world f) || (modelCheckKrpk model world g)
modelCheckKrpk (KMo relations valuations) world (Knows agent f)
                                     = case (lookup agent relations) of
                                          Just relation_a → case (lookup world relation_a) of
                                             Just neighbors → all (λx → modelCheckKrpk (KMo relations valuations) x (f) ) neighbors
                                             Nothing → True
                                          Nothing → True

kripkeToSuccint :: KripkeM → SuccEpistM
kripkeToSuccint (KMo relations valuations) = Mo ap beta programs
                                          where ap = sort (ap_M ++ ap_W)
                                                beta = make_beta ap_M ap_W valuations
                                                programs = [(agent, relationToProgram relation ap beta) | (agent, relation) ← relations]
                                                ap_M = nub $ concat [snd x | x ← valuations]
                                                ap_W = [-(fst x) | x ← valuations]
```

2.5 Model Translation Algorithms

```
succintToKripke :: SuccEpistM → KripkeM
succintToKripke (Mo ap beta programs) = KMo relations worldsAndAssignments
                                     where relations = [(agent, programToRelation program ap beta worldsAndAssignments) | (agent, program) ← programs]
                                           worldsAndAssignments = zip [1..(toInteger $ length assignments)] assignments
                                           assignments = [ assignment | assignment ← (allAssignmentsFor ap), satisfies assignment beta]

programToRelation :: Program → [Proposition] → Form → [(World, Assignment)] → Relation
programToRelation program ap beta worldsAndAssignments = programToRelation1 program ap beta worldsAndAssignments worldsAndAssignments

programToRelation1 :: Program → [Proposition] → Form → [(World, Assignment)] → [(World, Assignment)] → Relation
programToRelation1 _ _ _ [] _ = []
programToRelation1 program ap beta ((world,assignment):rest) worldsAndAssignments =
        (world, [vorld | (vorld, ass1) ← worldsAndAssignments, runProgram beta ap program assignment ass1]):(programToRelation1 program ap beta rest worldsAndAssignments)

relationToProgram :: Relation → [Proposition] → Form → Program
relationToProgram relation ap beta = relationToProgram1 relation ap

relationToProgram1 :: Relation → [Proposition] → Program
relationToProgram1 [] _ = IdleProgram
relationToProgram1 ((world, neighbors):rest) ap = Cup (relationToProgram2 world neighbors ap) (relationToProgram1 rest ap)

relationToProgram2 :: World → [World] → [Proposition] → Program
relationToProgram2 _ [] _ = IdleProgram
relationToProgram2 w_1 (w_2:rest) ap = Cup ( Semicolon (Semicolon (Question (P (-w_1)) ) UniversalProgram) (Question (P (-w_2))) ) (relationToProgram2 w_1 rest ap)
```

3 Model Checking Examples

```
my_kmodel = KMo
             [(1, [(1,[2,3])]),
                (2, [(2,[4,5]), (3,[6])])]
             [(4, [1,2]), (5,[1]), (6,[1,2]), (1,[]), (2,[]), (3,[])]

phi = Knows 1 (Knows 2 (P 2))
```

Below is the the encoding of the muddy children example without updates. All agents know that at least one of them is muddy, and each agent only knows the status of the other two.

```
my_kmodel2 = KMo
             [(1, [(1,[2]), (2,[1]), (3,[4]), (4,[5])]),
              (2, [(1,[3]), (3,[1]), (5,[6]), (6,[5])]),
             (3, [(1,[6]), (6,[1]), (2,[7]), (7,[2])])]
             [(1, [1,2, 3]), (2,[2,3]), (3,[1,3]), (4,[3]), (5,[1]), (6,[1,2]), (7,[2])]

phi2= Knows 1 (Knows 3 (Conj (P 1) (P 2)))

phi3 = Conj (Knows 1 (P 1)) (Knows 2 (P 2))
```

After model checking phi 1,2,3 on my_kmodel2 on all its worlds, on both the original Kripke model and its translation to a mental model, we have that both the kripke one and the mental one agree as expected (and the result is consistent with the usual semantics). The results are as follows:

```
*Main> modelCheckKrpk my_kmodel2 2 phi2
[True
*Main> modelCheckKrpk my_kmodel2 3 phi2
[True
*Main> modelCheckKrpk my_kmodel2 4 phi2
[True
*Main> modelCheckKrpk my_kmodel2 5 phi2
[True
*Main> modelCheckKrpk my_kmodel2 6 phi2
True
*Main> modelCheckKrpk my_kmodel2 7 phi2
True
*Main> modelCheckKrpk my_kmodel2 1 phi3
[False
*Main> modelCheckKrpk my_kmodel2 2 phi3
[True
*Main> modelCheckKrpk my_kmodel2 3 phi3
[False
*Main> modelCheckKrpk my_kmodel2 4 phi3
[True
*Main> modelCheckKrpk my_kmodel2 5 phi3
[True
*Main> modelCheckKrpk my_kmodel2 6 phi3
[False
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-1,1,2,3] phi
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-1,1,2,3] phi2
[False
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-1,1,2,3] phi3
[False
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-2,2,3] phi2
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-2,2,3] phi3
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-3,1,3] phi2
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-3,1,3] phi3
[False
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-4,3] phi2
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-4,3] phi3
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-5,1] phi2
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-5,1] phi3
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-6,1,2] phi2
[True
*Main> modelCheck (kripkeToSuccint my_kmodel2 ) [-6,1,2] phi3
[False
*Main>
```