Modal Logic Simulator

Iain Moncrief
School of EECS
Oregon State University

June 8, 2022

1 Introduction and Definitions

Epistemic modal logic is a subfield of modal logic that attempts to describe the reasoning of modalities such as knowledge and belief. Here, we narrow our focus to epistemic logics that are specifically concerned with knowledge, and the knowledge of entities that can know propositions (these entities are called *agents*). Even though our work is very specific to modal logics with a single modal operator for knowledge, our implementation is very general and can easily be extended to support other epistemic modalities, and even more general modalities. The languages of modal logics are parameterized by the atomic propositions which the propositional formulae will be built from, the operators which correspond to modalities, and the agents for which the modal operators relate to propositions via the modality.

1.1 Object Language

Given a set of atomic propositions At, a set of agents Ag, and a set of modal operators Op, the language of a modal logic is the set $\mathsf{L}(\mathsf{At}, \mathsf{Ag}, \mathsf{Op})$, defined recursively via

$$\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag}, \mathsf{Op}) ::= \; p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box\varphi$$

where $p \in \mathsf{At}$ and $\Box \in \mathsf{Op}$ are metavariables over atomic propositions and modal operators. Should the language be intended to describe modalities for multiple distinct agents, the language definition is modified to be

$$\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag}, \mathsf{Op}) ::= \; p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box_\alpha\varphi$$

where $\alpha \in \mathsf{Ag}$ is a metavariable ranging over the set of agents. Since we are only going to consider the knowledge modality, our set of modal operators is simply $\mathsf{Op} = \{K\}$, thus from here on, we abbreviate $\mathsf{L}(\mathsf{At}, \mathsf{Ag}, \{K\}) = \mathsf{L}(\mathsf{At}, \mathsf{Ag})$ our object language. For clarity, the language we reason about is $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$, which is defined via

$$\begin{gathered} \varphi \in L(\mathsf{At}, \mathsf{Ag}) ::= \; p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_\alpha\varphi \\ \alpha \in \mathsf{Ag} \qquad p \in \mathsf{At} \end{gathered}$$

parameterized by the set At of primitive propositions and the set Ag of agents.

The productions that define $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$ don't seem to completely resemble the syntax of what one would expect from a logic, but this representation (along with some intuitive assumptions) yields a simple foundation to define more expressive connectives. We define the abbreviations, for any sentence $\varphi, \psi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$ and primitive proposition $p \in \mathsf{At}$,

$$\begin{aligned} \bot &\equiv p \wedge \neg p \\ \top &\equiv \neg\bot \\ \varphi \vee \psi &\equiv \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \to \psi &\equiv \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi &\equiv (\varphi \to \psi) \wedge (\psi \to \varphi) \end{aligned}$$

thus introducing a shorthand for writing falsehood, tautology, disjunction, implication, and bidirectional implication.

1.2 Language Semantics

We define some very basic rules of inference that preserve the truth of formulas in $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$, which then allow us to define logical axioms more clearly. We have the following basic rules

$$\frac{}{\top}\ \textsc{Taut} \qquad \frac{\varphi}{\neg\neg\varphi}\ \neg\text{-}\textsc{Intro} \qquad \frac{\neg\neg\varphi}{\varphi}\ \neg\text{-}\textsc{Elim}$$

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}\ \wedge\text{-}\textsc{Intro} \qquad \frac{\psi \wedge \varphi}{\varphi \wedge \psi}\ \wedge\text{-}\textsc{Sym} \qquad \frac{\varphi \wedge \psi}{\varphi}\ \wedge\text{-}\textsc{Elim}$$

$$\frac{\varphi}{\varphi \vee \psi}\ \vee\text{-}\textsc{Intro} \qquad \frac{\psi \vee \varphi}{\varphi \vee \psi}\ \vee\text{-}\textsc{Sym} \qquad \frac{\varphi \vee \psi \quad \neg\varphi}{\psi}\ \vee\text{-}\textsc{Elim}$$

which allow us to define axioms within the language, which we discuss later. We have not verified that these are the minimal set of rules, but they provide the sufficient deductions for our language. We then have the axioms

$$K_\alpha(\varphi \wedge \psi) \leftrightarrow K_\alpha\varphi \wedge K_\alpha\psi\ K\text{-}\textsc{And} \qquad K_\alpha(\varphi \to \psi) \to K_\alpha\varphi \to K_\alpha\psi\ K\text{-MP}$$

$$K_\alpha\varphi \to \neg K_\alpha\neg\varphi\ K\text{-}\textsc{Valid} \qquad K_\alpha\varphi \to \varphi\ K\text{-}\textsc{Refl}$$

which characterize the knowledge modality.

1.2.1 Truth

Truth for non-epistemic proposition $\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$ (φ is absent of any modal operator) is determined by some valuation on At

$$V : \mathsf{At} \to \{\top, \bot\}$$

which is a function that maps a primitive proposition to a true or false value. Then we can have the rules

$$\frac{p \in \mathsf{At} \quad V(p) = \top}{p}\ \textsc{Eval-T} \qquad \frac{p \in \mathsf{At} \quad V(p) = \bot}{\neg p}\ \textsc{Eval-F}.$$

Simply put, for a non-epistemic proposition φ, given a valuation V, is true iff $\varphi[p \mapsto V(p)]$ is true, using the standard conjunction and negation truth tables, applied inductively on the expression. A desirable representation of the semantics of $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$ would be to have a semantic function

$$[\![\cdot]\!] : \mathsf{L}(\mathsf{At}, \mathsf{Ag}) \to (\mathsf{At} \to \{\top, \bot\}) \to \{\top, \bot\}$$

where the semantic domain is the set of all valuations on At. However more is needed to appropriately decide truth for propositions that contain modal operators.

1.2.2 Kripke Models

Kripke Models are structures that are used to determine truth for formulae in modal logics. For an epistemic logic $\mathsf{L}(\mathsf{At}, \mathsf{Ag}, \mathsf{Op})$, a Kripke Model is a 3-tuple $M = \langle S, R^{\mathsf{Ag}}, V^{\mathsf{At}} \rangle$, where

- S is a nonempty set of states/worlds,
- $R^{\mathsf{Ag}} : \mathsf{Ag} \to \mathcal{P}(S \times S)$ is a function that maps an agent to a relation on states,
- $V^{\mathsf{At}} : S \to \mathsf{At} \to \{\top, \bot\}$ is a function that maps each state to a valuation on At.

We abbreviate acc_α to be the accessibility relation $R^{\mathsf{Ag}}(\alpha)$ for $\alpha \in \mathsf{Ag}$, and val_s to be the valuation function for some state $s \in S$.

Given a pair (M, s) of a Kripke Model and a state, we can determine if a sentence $\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$ is true, which we express via the notation $(M, s) \models \varphi$. We define this relation for $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$ as

$$\begin{aligned}
(M, s) \models p &\iff \mathsf{val}_s(p) = \top \\
(M, s) \models \neg\varphi &\iff (M, s) \not\models \varphi \\
(M, s) \models \varphi \wedge \psi &\iff (M, s) \models \varphi \text{ and } (M, s) \models \psi \\
(M, s) \models K_\alpha \varphi &\iff \forall (s, t) \in \mathsf{acc}_\alpha : (M, t) \models \varphi.
\end{aligned}$$

where the rule for the knowledge modality can be thought of intuitively as: *agent α knows φ in the world s if in all worlds α considers, φ is true.*[1]

2 Implementation

2.1 Type Representations

When formulating the representations used in our implementation, we made a point to preserve the generality of the formal definitions and avoided deviating for sake of efficiency.

For our representation of the language $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$, parameterized by the set of atomic propositions At and the set of agents Ag, we use a parametric data type

```
data L agent prim
  = Prim prim
  | Neg (L agent prim)
  | And (L agent prim) (L agent prim)
  | Know agent (L agent prim)
```

where each production rule of $\mathsf{L}(\mathsf{At}, \mathsf{Ag})$ corresponds to a constructor of `L agent prim` parameterized by the type of agents `agent` and the type of atomic propositions `prim`.

We use lists to represent general collections, which are intended to resemble sets, but using the `Set` container in Haskell imposes typeclass constraints on functions that are intended to be as general as possible. We have the type synonym

```
type Collection a = [a]
```

The type representation of a Kripke Model is defined using the types for accessibility relations and valuation functions, represented as

```
type AccessRelation agent state = agent -> Collection (state,state)
type Valuation state prim = state -> prim -> Bool
```

where `state` is the type of states. Now, we define the type of a Kripke Model to be the data type

```haskell
data KripkeModel agent prim state
  = M { agents        :: Collection agent
      , prims         :: Collection prim
      , states        :: Collection state
      , accessibility :: AccessRelation agent state
      , valuation     :: Valuation state prim
      }
```

parameterized by the type of states `state`, type of atomic propositions `prim`, and the type of agents `agent`.

While the type definition for Kripke Models is very general, most of the implementation's core features are implemented using a more specific type, namely

```haskell
type State prim = Collection prim
type KripkeModel' agent prim = KripkeModel agent prim (State prim)
```

where `Eq prim` is assumed. This allows us to have clean and efficient implementations of functions that construct Kripke Models. Motivation is discussed in section 4.

2.2 Essence

The core of our implementation is made up of three functions: Given a sentence $\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$,

```haskell
genKripke :: L agent prim -> (KripkeModel agent prim (State prim),[State prim])
```

will produce a pair of a Kripke Model M and a set of satisfying states $S' \subseteq S$ such that $\forall s \in S' : (M, s) \models \varphi$. Given a Kripke Model M, a state s, and a sentence $\varphi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$,

```haskell
models :: KripkeModel agent prim state -> state -> L agent prim -> Bool
```

will decide if $(M, s) \models \varphi$. Given the sentences $\varphi, \psi \in \mathsf{L}(\mathsf{At}, \mathsf{Ag})$,

```haskell
entails :: L agent prim -> L agent prim -> Bool
entails p p' = and [models km s p' | s <- ss]
    where (km,ss) = genKripke p
```

will decide if ψ is entailed by φ.

2.3 Usage

Our implementation can be used to validate satisfyability, decided semantic entailment of a model, solve for models that entail a sentence, and determine if a formula follows from certain assumptions. Additionally, MLS can be applied to problems in a way that isolates the necessary encodings.

Using the function,

```haskell
(|=) :: (KripkeModel agent prim state, state) -> L agent prim -> Bool
(|=) (km,s) phi = models km s phi
```

users may test sentences against their own Kripke Model that they have defined, via some situation resembling

```
type Agent = Char
type Prim  = Int
type State = Map Int Bool
testModel :: KripkeModel Agent Prim State
testState :: State
formula :: L Agent Prim
formula = Know 'a' (Prim 1 `And` Prim 2)
```

then evaluating `(testModel,testState) |= formula` will decide if the model-state pair semantically entails the formula. Users will most likely find it undesirable to have to encode each knowledge structure they want to analyze as a value in Haskell. The function `genKripke` will produce a model of a given sentence, which makes it much easier to analyze propositions. It is conceivable that users may want to avoid handling Kripke Models at all, and just test logical entailment. This is done using the `entails` function, which does not concern the user with how truth of epistemic sentences is represented.

Other interesting functions can be defined, such as

```
agentKnowledge :: L agent prim -> agent -> [L agent prim]
```

which given a sentence and an agent, yields a list of propositions that the agent knows. Another example is the funnction

```
satStates :: KripkeModel agent prim state -> L agent prim -> [state]
```

that returns the states in a Kripke Model where a formula holds.

3 Outcomes

Here we examine how our implementation behaves for a basic example.

Let us analyze the models and consequences of the statements: *"Agent 'a' knows (proposition 1 and proposition 2).",* and *"Agent 'b' knows proposition 2.".* We can write these formally as

$$K_a(p_1 \wedge p_2) \qquad K_b\ p_2.$$

Now let $\varphi \equiv K_a(p_1 \wedge p_2) \wedge K_b\ p_2$ be the conjunction of the two. We can then say $\varphi \in \mathsf{L}(\{p_1, p_2\}, \{a, b\})$, then it makes sense to have apply the inference rules, resulting in the deductions

$$\dfrac{\dfrac{\dfrac{K_a(p_1 \wedge p_2) \wedge K_b\ p_2}{K_a(p_1 \wedge p_2)}\ \wedge\text{-}\textsc{Elim}}{K_a\ p_1 \wedge K_a\ p_2}\ K\text{-}\textsc{And}}{K_a\ p_1}\ \wedge\text{-}\textsc{Elim} \qquad \dfrac{\dfrac{\dfrac{K_a(p_1 \wedge p_2) \wedge K_b\ p_2}{K_a(p_1 \wedge p_2)}\ \wedge\text{-}\textsc{Elim}}{K_a\ p_1 \wedge K_a\ p_2}\ K\text{-}\textsc{And}}{K_a\ p_2}\ \wedge\text{-}\textsc{Elim}$$

$$\dfrac{\dfrac{K_a(p_1 \wedge p_2) \wedge K_b\ p_2}{K_b\ p_2}\ \wedge\text{-}\textsc{Elim}}{\neg K_b \neg p_2}\ K\text{-}\textsc{Valid}$$

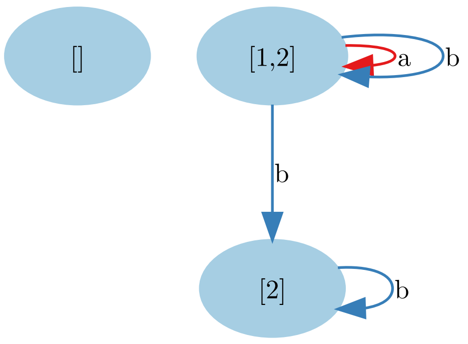
as well as $\neg K_b\ p_1$, since p_1 is clearly not in b's knowledge base. Now we test our implementation by representing these propositions as AST values

```
phi = Know 'a' (Prim 1 `And` Prim 2) `And` Know 'b' (Prim 2)
tests = [ Know 'a' (Prim 1)
        , Know 'a' (Prim 2)
        , Neg $ Know 'b' $ Neg $ Prim 2
        , Neg $ Know 'b' (Prim 1) ]
```

We can then obtain a Kripke Model that satisfies `formula`, via the evaluation

```
genKripke phi == (km,[[1,2]])
```

We can then visualize `km` as a directed graph,



$K_a(p_1 \wedge p_2) \wedge K_b\ p_2$

then running each test against the model, we have

```
map ((km,[1,2])|=) tests == [True,True,True,True]
```

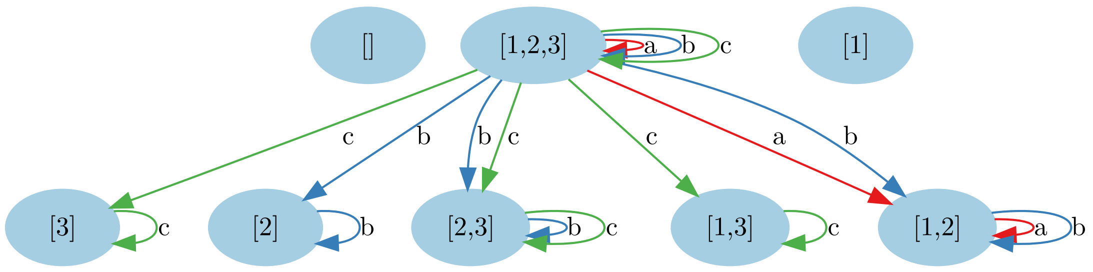
as expected. We can also run

```
map (entails phi) tests == [True,True,True,True]
```

which evaluates as expected. If we modify `phi` to have an additional clause of the knowledge of an agent c,

```
phi' = Know 'a' (Prim 1 `And` Prim 2) `And` Know 'b' (Prim 2) `And` Know 'c' (Prim 3)
```

we can see how the Kripke Model changes and becomes much more complicated



$K_a(p_1 \wedge p_2) \wedge K_b\ p_2 \wedge K_c\ p_3$

since a new agent and primitive proposition was introduced.

These operations would be useful to aid researchers and as an educational tool.

4 Design Evolution and Future Work

The datatype definition for Kripke Models was articulated early on in the project, though the best definition for `Valuation state prim` was unclear. The formal definition of a valuation function `val` for a Kripke Model is the assignment of a truth mapping to each state in the Kripke Model. A correct, naive implementation would be

```
type TMapping a = Map a Bool
type Valuation state prim = Map state (TMapping prim)
```

This implementation is correct, but would require the whole valuationn relation to be held in memory when used, thus would not leverage lazy evaluation. It would also impose considerable typeclass constraints on other functions, due to the `Ord` typeclass requirements needed by most of the functions in the `Data.Map` library. Then it was considered to have the states of a KripkeModel be the truth valuationns themselves, so

```
type State prim = [(prim,Bool)]
type Valuation prim = State prim -> prim -> Bool
```

then the definition of the valuation function would come nicely as

```
valuation :: Eq prim => Valuation prim
valuation = flip lookup
```

but still requires a typeclass constraint and doesn't allow for much lazieness. Finally, an efficient representation for states and valuations was decided on. It was realized that each state $s \in S$ could be represented as an element of the powerset of primitive propositios $s \in \mathcal{P}(\mathsf{At})$, so then the valuation function becomes equivelent to set inclusion $\mathsf{val}_s(p) \iff p \in s \quad \forall p \in \mathsf{At}$. So then the types follow

```
type State prim = Collection prim
type KripkeModel' agent prim = KripkeModel agent prim (State prim)
```

then we have

```
decide :: Eq prim => State prim -> prim -> Bool
decide = flip elem
consKM :: (Eq agent, Eq prim) =>
          [agent] ->
          [prim] ->
          AccessRelation agent (State prim) ->
          KripkeModel' agent prim
consKM ags pps access
  = M { agents        = ags
      , prims         = pps
      , states        = subsets pps
      , valuation     = decide
      , accessibility = access
      }
```

which makes it much easier to construct Kripke Models in more general ways, without having to define total valuation functions.

The core functions are composed of the more basic operations

```
satState :: Valuation state prim -> state -> L agent prim -> Bool
agentKnowledge :: Eq agent => L agent prim -> agent -> [L agent prim]
agentsUsed :: Eq ag => L ag at -> [ag]
primsUsed  :: Eq at => L ag at -> [at]
```

which then let us define

```
satStates :: Valuation state at -> [state] -> L ag at -> [state]
satStatesKM :: KripkeModel ag at state -> L ag at -> [state]
```

and just with these functions, we have implemented an efficient algorithm for finding satisfying Kripke Models for a given formula.

Due to the accessibility relation type definition, there was an unintended consequence in our implementation, which is the constraint `Eq state` becomes necessary when deciding $(M, s) \models K_a\varphi$. The corresponding expression has the trace

```
(km,s) |= (Know a phi)
  = and [(km,t) |= phi | (s',t) <- accessibility km a
                       , s == s']
```

and `accessibility km a` yields a general relation, and to distinguish just the pairs $(s', t) \in \mathtt{acc}_a$ such that $s = s'$, a form of equality is needed. Future implementations could yield more generic types. One solution would be to modify the type

```
type AccessRelation agent state = agent -> Collection (state,state)
```

to the isomorphic type

```
type AccessRelation agent state = agent -> state -> Collection state
```

which is would avoid the need for equality inside the `models` function

```
(km,s) |= (Know a phi) = and [(km,t) |= phi | t <- accessibility km a s]
```

thus removing the need for any typeclass constraint.

In the future, the language data type could be generalized to support other modal operators, by

```
data L op ag at
  = Prim at
  | Neg (L op ag at)
  | And (L op ag at) (L op ag at)
  | Modal op ag (L op ag at)
```

where the language type `L op ag at` is parameterized by the type of modal operators `op`. Also, it would be pretty to have a general way to export general Kripke Models as DOT files for visualization.

References

[1] Hans van Ditmarsch. *Handbook of Epistemic Logic*. College Publications, 2015.