$\mathcal{KL}$ as a Knowledge Base Logic in Haskell

Natasha De Kriek, Milan Hartwig, Victor Joss, Paul Weston, Louise Wilk

Tuesday 25th March, 2025

Abstract

In this project, we aim to implement the first-order epistemic logic $\mathcal{KL}$ as introduced by Levesque (1981) and refined by Levesque and Lakemeyer (2001). The semantics for this logic evaluates formulae on world states and epistemic states where world states are sets of formulae that are true at the world and epistemic states are sets of world states that are epistemically accessible. Levesque and Lakemeyer use the language $\mathcal{KL}$ as "a way of communicating with a knowledge base" (ibid. p. 79). For this, they define an ASK- and a TELL-operation on a knowledge base. In our project, we implement a $\mathcal{KL}$-model, the ASK- and TELL- operations, a tableau-based satisfiablity and validity checking for $\mathcal{KL}$, as well as compare $\mathcal{KL}$-models to epistemic Kripke models and implement a translation function between them.

Contents

1 $\mathcal{KL}$: Syntax and Semantics

1.1 Syntax of $\mathcal{KL}$

The syntax of the language $\mathcal{KL}$ is described in Levesque and Lakemeyer (2001) and inspired by Levesque's work (Levesque 1981). The SyntaxKL module establishes the foundation for $\mathcal{KL}$'s syntax, defining the alphabet and grammar used in subsequent semantic evaluation.

```
{-# LANGUAGE InstanceSigs #-}


module SyntaxKL where

import Test.QuickCheck
```

Symbols of $\mathcal{KL}$

The expressions of $\mathcal{KL}$ are constituted by sequences of symbols drawn from the following two sets (cf. Levesque 1981): Firstly, the *logical symbols*, which consist of the logical connectives and quantifiers $\exists, \vee, \neg$, as well as punctuation and parentheses. Furthermore, it compromises a countably infinite supply of first-order variables denoted by the set $\{x, y, z, \ldots\}$, a countably infinite supply of standard names, represented by the set $\{\#1, \#2, \ldots\}$, and the equality symbol $=$. The *non-logical symbols* comprise predicate symbols of any arity $\{P, Q, R, \ldots\}$, which are intended to represent domain-specific properties and relations, and function symbols of any arity, which are used to denote mappings from individuals to individuals (Levesque and Lakemeyer 2001, p.22).

n this implementation, standard names are represented as strings (e.g., "n1", "n2") via the StdName type, and variables are similarly encoded as strings (e.g., "x", "y") with the Variable type, ensuring that we have a distinct yet infinite supplies of each.

```
arbitraryUpperLetter :: Gen String
arbitraryUpperLetter = (:[]) <$> elements ['A'..'Z']

arbitraryLowerLetter :: Gen String
arbitraryLowerLetter = (:[]) <$> elements ['a'..'z']

-- Represents a standard name (e.g., "n1") from the infinite domain N
newtype StdName = StdName String deriving (Eq, Ord, Show)
instance Arbitrary StdName where
  arbitrary:: Gen StdName
  arbitrary = StdName . ("n" ++) . show <$> elements [1 .. 20::Int]

-- Represents a first-order variable (e.g., "x")
newtype Variable = Var String deriving (Eq, Ord, Show)
instance Arbitrary Variable where
  arbitrary:: Gen Variable
  arbitrary = Var . show <$> elements [1 .. 20::Int]
```

Terms and Atoms

Terms in $\mathcal{KL}$ are the building blocks of expressions, consisting of variables, standard names, or function applications. Atomic propositions (atoms) are formed by applying predicate symbols to lists of terms. To distinguish primitive terms (those that contain no variable and only a single function symbol) and primitive atoms (those atoms that contain no variables and only standard names as terms) for semantic evaluation, we also define PrimitiveTerm and PrimitiveAtom.

```
-- Defines terms: variables, standard names, or function applications
data Term = VarTerm Variable    -- A variable (e.g., "x")
          | StdNameTerm StdName    -- A standard name (e.g., "n1")
          | FuncAppTerm String [Term]    -- Function application (e.g., "Teacher" ("x"))
          deriving (Eq, Ord, Show)
```

```
instance Arbitrary Term where
  arbitrary :: Gen Term
  arbitrary = sized $ \n -> genTerm (min n 5) where
    genTerm 0 = oneof [VarTerm <$> arbitrary,
                       StdNameTerm <$> arbitrary]
    genTerm n = oneof [VarTerm <$> arbitrary,
                       StdNameTerm <$> arbitrary,
                       FuncAppTerm <$> arbitraryLowerLetter
                                   <*> resize (n `div` 2) (listOf1 (genTerm (n `div` 2)))]

-- Terms with no variables and only a single function symbol
data PrimitiveTerm = PStdNameTerm StdName   -- e.g., "n1"
                   | PFuncAppTerm String [StdName]
  deriving (Eq, Ord, Show)

-- Define Atoms as predicates applied to terms
data Atom = Pred String [Term]  --e.g. "Teach" ("n1", "n2")
  deriving (Eq, Ord, Show)

instance Arbitrary Atom where
  arbitrary :: Gen Atom
  arbitrary = sized $ \n -> genAtom (min n 5) where
      genAtom :: Int -> Gen Atom
      genAtom 0 = Pred <$> arbitraryLowerLetter <*> pure []
      genAtom n = Pred <$> arbitraryLowerLetter <*> vectorOf n arbitrary

-- Atoms with only standard names as terms
data PrimitiveAtom = PPred String [StdName]
  deriving (Eq, Ord, Show)
```

Formulas

$\mathcal{KL}$-formulas are constructed recursively from atoms, equality, and logical operators. The Formula type includes atomic formulas, equality between terms, negation, disjunction, existential quantification, and the knowledge operator K. Additional connectives like universal quantification ($\forall$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$) are defined as derived forms for convenience.

```
--Defines KL-formulas with logical and epistemic constructs
data Formula = Atom Atom                -- Predicate (e.g. Teach(x, "n1"))
             | Equal Term Term          -- Equality (e.g., x = "n1")
             | Not Formula              -- Negation
             | Or Formula Formula       -- Disjunction
             | Exists Variable Formula  -- Existential (e.g., exists x (Teach x "sue"))
             | K Formula                -- Knowledge Operator (e.g., K (Teach "ted" "sue"))
             deriving (Eq, Ord, Show)

instance Arbitrary Formula where
  arbitrary :: Gen Formula
  arbitrary = sized $ \n -> genFormula (min n 5)   where
    genFormula 0 = oneof [Atom <$> arbitrary,
                          Equal <$> arbitrary <*> arbitrary]
    genFormula n = oneof [Not <$> genFormula (n `div` 2),
                          Or <$> genFormula (n `div` 2) <*> genFormula (n `div` 2),
                          Exists <$> arbitrary <*> genFormula (n `div` 2),
                          K <$> genFormula (n `div` 2)]

-- Universal quantifier as derived form
klforall :: Variable -> Formula -> Formula
klforall x f = Not (Exists x (Not f))

-- Implication as derived form
implies :: Formula -> Formula -> Formula
implies f1 = Or (Not f1)

-- Biconditional as derived form
iff :: Formula -> Formula -> Formula
iff f1 f2 = Or (Not (Or f1 f2)) (Or (Not f1) f2)
```

We can now use this implementation of $\mathcal{KL}$'s syntax to implement the semantics.

1.2 Semantics of $\mathcal{KL}$

$\mathcal{KL}$ is an epistemic extension of first-order logic designed to model knowledge and uncertainty, as detailed in Levesque and Lakemeyer (2001). It introduces a knowledge operator K and uses an infinite domain $\mathcal{N}$ of standard names to denote individuals. Formulas are evaluated in world states: consistent valuations of atoms and terms, while epistemic states capture multiple possible worlds, reflecting epistemic possibilities.
The semantics are implemented in the SemanticsKL module, which imports syntactic definitions from SyntaxKL and uses Haskell's Data.Map and Data.Set for efficient and consistent mappings.

```
{-# LANGUAGE InstanceSigs #-}

module SemanticsKL where

import SyntaxKL
import Data.Map (Map)
import qualified Data.Map as Map
import Data.Set (Set)
import qualified Data.Set as Set

import Test.QuickCheck
```

Worlds and Epistemic States
A WorldState represents a single possible world in $\mathcal{KL}$, mapping truth values to primitive atoms and standard names to primitive terms. An EpistemicState, defined as a set of WorldStates, models the set of worlds an agent considers possible, enabling the evaluation of the K operator.

```
-- A single world state with valuations for atoms and terms
data WorldState = WorldState
 { atomValues :: Map Atom Bool,     -- Maps (primitive) atoms to truth values
   termValues :: Map Term StdName    --Maps (primitive) terms to standard names
  }  deriving (Eq, Ord, Show)

instance Arbitrary WorldState where
  arbitrary :: Gen WorldState
  arbitrary = WorldState <$> arbitrary <*> arbitrary

-- A set of possible world states, modeling epistemic possibilities
type EpistemicState = Set WorldState
```

Constructing World States
We can construct world states by using mkWorldState, which builds a WorldState from lists of primitive atoms and terms. While a WorldState is defined in terms of Atom and Term, we use mkWorldState to make sure that we can only have primitive atoms and primitive terms in the mapping. To be able to use primitive terms and atoms in other functions just as we would use atoms and terms (since primitive atoms and primitive terms are atoms and terms as well), we convert the constructors to those of regular terms and atoms. We then use the function checkDups to ensure that there are no contradictions in the world state (e.g., P(n1) mapped to both True and False), thus reinforcing the single-valuation principle (Levesque and Lakemeyer 2001, p. 24). mkWorldState then constructs maps for efficient lookup.

```

-- Constructs a WorldState from primitive atoms and primitive terms
mkWorldState :: [(PrimitiveAtom, Bool)] -> [(PrimitiveTerm, StdName)] -> WorldState
mkWorldState atoms terms =
  let convertAtom (PPred p ns, b) = (Pred p (map StdNameTerm ns), b)  -- Convert primitive
      atom to Atom
```

```haskell
      convertTerm (PStdNameTerm n, v) = (StdNameTerm n, v)  -- Convert primitive term to
          Term
      convertTerm (PFuncAppTerm f ns, v) = (FuncAppTerm f (map StdNameTerm ns), v)
      atomList = map convertAtom atoms
      termList = map convertTerm terms
  in WorldState (Map.fromList (checkDups atomList)) (Map.fromList (checkDups termList))

-- Checks for contradictory mappings in a key-value list
checkDups :: (Eq k, Show k, Eq v, Show v) => [(k, v)] -> [(k, v)]
checkDups [] = []  -- Empty list is consistent
checkDups ((k, v) : rest) =  -- Recursively checks each key k against the rest of the list.
  case lookup k rest of
    Just v' | v /= v' -> error $ "Contradictory mapping for " ++ show k ++ ": " ++ show v
        ++ " vs " ++ show v' -- If k appears with a different value v', throws an error.
    _ -> (k, v) : checkDups rest  -- Keep pair if no contradiction
```

Since we have decided to change the constructors of data of type PrimitiveAtom or PrimitiveTerm to those of Atom and Term, we have implemented two helper-functions to check if a Term or an Atom is primitive. This way, we can, if needed, check whether a given term or atom is primitive and then change the constructors appropriately.

```haskell
-- Checks if a term is primitive (contains only standard names)
isPrimitiveTerm :: Term -> Bool
isPrimitiveTerm (StdNameTerm _) = True
isPrimitiveTerm (FuncAppTerm _ args) = all isStdName args
  where isStdName (StdNameTerm _) = True
        isStdName _ = False
isPrimitiveTerm _ = False

-- Checks if an atom is primitive
isPrimitiveAtom :: Atom -> Bool
isPrimitiveAtom (Pred _ args) = all isStdName args
  where isStdName (StdNameTerm _) = True
        isStdName _ = False
```

Term Evaluation

To evaluate a ground term in a world state, we define a function evalTerm that takes a WorldState and a Term and returns a StdName. The idea is to map syntactic terms to their semantic values (standard names) in a given world state. The function uses pattern matching to handle the three possible forms of Term:

1. VarTerm _
   If the term is a variable (e.g., x), it throws an error. This enforces a precondition that evalTerm only works on ground terms (terms with no free variables). In $\mathcal{KL}$, variables must be substituted with standard names before evaluation, aligning with the semantics where only ground terms have denotations (Levesque and Lakemeyer 2001, p. 24). This is a runtime check to catch ungrounded inputs.

2. StdNameTerm n
   If the term is a standard name wrapped in StdNameTerm (e.g., StdNameTerm (StdName "n1")), it simply returns the underlying StdName (e.g., StdName "n1"). Standard names in $\mathcal{KL}$ are constants that denote themselves (ibid., p.22). For example, if n=StdName "n1", it represents the individual n1, and its value in any world is n1. In this case, no lookup or computation is needed.

3. FuncAppTerm f args
   If the term is a function application (e.g., f(n1,n2)), evalTerm evaluates the argument, by recursively computing the StdName values of each argument in args using evalTerm

w. Next, the ground term is constructed: It Builds a new FuncAppTerm term where all arguments are standard names (wrapped in StdNameTerm), ensuring it's fully ground. We then look up the value by querying the termValues map in the world state w for the denotation of this ground term, erroring on undefined terms.

```
-- Evaluates a ground term to its standard name in a WorldState
evalTerm :: WorldState -> Term -> StdName
evalTerm w t = case t of
  VarTerm _ -> error "evalTerm: Variables must be substituted"  -- Variables are not ground
  StdNameTerm n -> n   -- Standard names denote themselves
  FuncAppTerm f args ->
    let argValues = map (evalTerm w) args   -- Recursively evaluate arguments
        groundTerm = FuncAppTerm f (map StdNameTerm argValues)   -- Construct ground term
    in case Map.lookup groundTerm (termValues w) of
        Just n -> n   -- Found in termValues
        Nothing -> error $ "evalTerm: Undefined ground term " ++ show groundTerm   -- Error
             if undefined
```

Groundness and Substitution

To support formula evaluation, isGround and isGroundFormula check for the absence of variables, while substTerm and subst perform substitution of variables with standard names, respecting quantifier scope to avoid capture. We need these functions to be able to define a function that checks whether a formula is satisfiable in a worldstate and epistemic state.

```
-- Check if a term is ground (contains no variables).
isGround :: Term -> Bool
isGround t = case t of
  VarTerm _ -> False
  StdNameTerm _ -> True
  FuncAppTerm _ args -> all isGround args

-- Check if a formula is ground.
isGroundFormula :: Formula -> Bool
isGroundFormula f = case f of
  Atom (Pred _ terms) -> all isGround terms
  Equal t1 t2 -> isGround t1 && isGround t2
  Not f' -> isGroundFormula f'
  Or f1 f2 -> isGroundFormula f1 && isGroundFormula f2
  Exists _ _ -> False   -- always contains a variable
  K f' -> isGroundFormula f'

-- Substitute a variable with a standard name in a term.
substTerm :: Variable -> StdName -> Term -> Term
substTerm x n t = case t of
  VarTerm v | v == x -> StdNameTerm n -- Replace variable with name
  VarTerm _ -> t
  StdNameTerm _ -> t
  FuncAppTerm f args -> FuncAppTerm f (map (substTerm x n) args)

-- Substitute a variable with a standard name in a formula.
subst :: Variable -> StdName -> Formula -> Formula
subst x n formula = case formula of
  Atom (Pred p terms) -> Atom (Pred p (map (substTerm x n) terms))
  Equal t1 t2 -> Equal (substTerm x n t1) (substTerm x n t2)
  Not f -> Not (subst x n f)
  Or f1 f2 -> Or (subst x n f1) (subst x n f2)
  Exists y f | y == x -> formula -- Avoid capture
             | otherwise -> Exists y (subst x n f)
  K f -> K (subst x n f)
```

Model and Satisfiability

Since we want to check for satisfiability in a model, we want to make the model explicit:

```
-- Represents a model with an actual world, epistemic state, and domain
data Model = Model
```

```
  { actualWorld :: WorldState       -- The actual world state
  , epistemicState :: EpistemicState -- Set of possible world states
  , domain :: Set StdName           -- Domain of standard names
  } deriving (Show)

instance Arbitrary Model where
  arbitrary:: Gen Model
  arbitrary = Model <$> arbitrary <*> arbitrary <*> arbitrary
```

A Model encapsulates an actual world, an epistemic state, and a domain, enabling the evaluation of formulas with the Koperator. satisfiesModel implements $\mathcal{KL}$'s satisfaction relation, checking truth across worlds.

```
-- Checks if a formula is satisfied in a model
satisfiesModel :: Model -> Formula -> Bool
satisfiesModel m = satisfies (epistemicState m) (actualWorld m)
  where
    satisfies e w formula = case formula of
      Atom (Pred p terms) ->
        if all isGround terms
          then Map.findWithDefault False (Pred p terms) (atomValues w)   -- Default False
              for undefined atoms
          else error "Non-ground atom in satisfies!"
      Equal t1 t2 ->
        if isGround t1 && isGround t2   -- Equality of denotations
          then evalTerm w t1 == evalTerm w t2
          else error "Non-ground equality in satisfies!"
      Not f ->
        not (satisfies e w f)
      Or f1 f2 ->
        satisfies e w f1 || satisfies e w f2
      Exists x f ->
      -- \(e, w \models \exists x. \alpha\) iff for some name \(n\), \(e, w \models \
          alpha_n^x\)
        any (\n -> satisfies e w (subst x n f)) (Set.toList $ domain m)
      -- \(e, w \models K \alpha\) iff for every \(w' \in e\), \(e, w' \models \alpha\)
      K f ->
        all (\w' -> satisfies e w' f) e
```

Grounding and Model Checking

Building on this we can implement a function checkModel that checks whether a formula holds in a given model. checkModel ensures a formula holds by grounding it with all possible substitutions of free variables, using groundFormula and freeVars to identify and replace free variables systematically.

```
-- Checks if a formula holds in a model by grounding it
checkModel :: Model -> Formula -> Bool
checkModel m phi = all (satisfiesModel m) (groundFormula phi (domain m))
```

Note that we use the function groundFormula here. Since we have implemented satisfiesModel such that it assumes ground formulas or errors out, we decided to handle free variables by grounding formulas, given a set of free standard names to substitute. Alternatives, would be to error or always substitute the same standard name. The the implementation that we have chosen is more flexible and allows for more varied usage, however it is computationally expensive (We would appreciate it if you have suggestions to improve this). We implement groundFormula as follows:

```

-- Generates all ground instances of a formula
groundFormula :: Formula -> Set StdName -> [Formula]
groundFormula f dom = groundFormula' f >>= groundExists dom
  where
    -- Ground free variables at the current level
    groundFormula' formula = do
      let fvs = Set.toList (freeVars formula)
```

```
    subs <- mapM (\_ -> Set.toList dom) fvs
    return $ foldl (\acc (v, n) -> subst v n acc) formula (zip fvs subs)

  -- Recursively eliminate Exists in a formula
  groundExists domainEx formula = case formula of
    Exists x f' -> map (\n -> subst x n f') (Set.toList domainEx) >>= groundExists
        domainEx
    Atom a -> [Atom a]
    Equal t1 t2 -> [Equal t1 t2]
    Not f' -> map Not (groundExists domainEx f')
    Or f1 f2 -> do
      g1 <- groundExists domainEx f1
      g2 <- groundExists domainEx f2
      return $ Or g1 g2
    K f' -> map K (groundExists domainEx f')
```

This function takes a formula and a domain of standard names and returns a list of all possible ground instances of the formula by substituting its free variables with elements from the domain. We use a function variables that identifies all the variables in a formula that need grounding or substitution. If the Boolean 'includeBound' is 'True', variables returns all variables (free and bound) in the formula. If 'includeBound' is 'False', it returns only free variables, excluding those bound by quantifiers. This way, we can use the function to support both 'freeVars' (free variables only) and 'allVariables' (all variables).

```
-- Collects variables in a formula, with a flag to include bound variables
variables :: Bool -> Formula -> Set Variable
variables includeBound = vars
  where
    -- Helper function to recursively compute variables in a formula
    vars formula = case formula of
      -- Union of variables from all terms in the predicate
      Atom (Pred _ terms) -> Set.unions (map varsTerm terms)
      -- Union of variables from both terms in equality
      Equal t1 t2 -> varsTerm t1 `Set.union` varsTerm t2
      Not f' -> vars f'
      Or f1 f2 -> vars f1 `Set.union` vars f2
      Exists x f' -> if includeBound
                     then Set.insert x (vars f') -- Include bound variable x if
                         includeBound is True
                     else Set.delete x (vars f') -- Exclude bound variable x if
                         includeBound is False
      K f' -> vars f'  -- Variables in the subformula under K (no binding)

    varsTerm term = case term of
      VarTerm v -> Set.singleton v  -- A variable term contributes itself to the set
      StdNameTerm _ -> Set.empty  -- A standard name has no variables
      FuncAppTerm _ args -> Set.unions (map varsTerm args)  -- Union of variables from all
          function arguments

-- Collects free variables in a formula
freeVars :: Formula -> Set Variable
freeVars = variables False

-- Collects all variables (free and bound) in a formula
allVariables :: Formula -> Set Variable
allVariables = variables True
```

2 Ask and Tell Operators

To use $\mathcal{KL}$ to interact with a knowledge base, Levesque and Lakemeyer (2001) defines two operators on epistemic states: ASK and $TELL$. Informally, ASK is used to determine if a sentence is known to a knowledge base whereas $TELL$ is used to add a sentence to the knowledge base. Since epistemic states are sets of possible worlds, the more sentences that are known reduces the set of possible worlds. For this purpose, an $INITIAL$ epistemic state is also defined to contain all possible worlds given a finite set of atoms and terms.

```
module AskTell (ask,askModel , tell , tellModel , initial) where
import qualified Data.Set as Set
import SyntaxKL
import SemanticsKL
```

The ASK operator determines whether a formula is known to a knowledge base. Formally, given an epistemic state e and any sentence α of $\mathcal{KL}$,

$$ASK[e, \alpha] = \begin{cases} True & \text{if } e \models K\alpha \\ False & \text{otherwise} \end{cases}$$

When implementing ASK in Haskell, we must take into account that a domain is implied by "$\models$" so that we can evaluate sentences with quantifiers. As such, we will take a domain as our first argument.

```
-- ASK (Definition 5.2.1)
ask :: Set.Set StdName -> EpistemicState -> Formula -> Bool
ask d e alpha | Set.null e = False
              | otherwise = satisfiesModel newModel (K alpha) where
              newModel = Model {actualWorld = (Set.findMin e), epistemicState = e, domain =
                  d}
```

We can simplify this into an askModel function that takes only a model and sentence as input.

```
askModel :: Model -> Formula -> Bool
askModel m alpha |  Set.null (epistemicState m) = False
             |  otherwise = satisfiesModel m (K alpha)
```

The second operation, $TELL$, asserts that a sentence is true and in doing so reduces which worlds are possible. In practice, $TELL[\varphi, e]$ filters the epistemic state e to worlds where the sentence φ holds. That is,

$$TELL[\varphi, e] = e \cap \{w \mid w \models \varphi\}$$

Again, we run into the issue that "$\models$" requires a domain, and so a domain must be specified to evaluate sentences with quantifiers.

```
-- TELL operation (Definition 5.5.1)
tell :: Set.Set StdName -> EpistemicState -> Formula -> EpistemicState
tell d e alpha = Set.filter filterfunc e where
    filterfunc = (\w -> satisfiesModel (Model {actualWorld = w, epistemicState = e, domain
        = d}) alpha)
```

We can again simplify to a function tellModel, that takes as input a model and formula and produces a model with a modified epistemic state.

```
tellModel :: Model -> Formula -> Model
tellModel m alpha = Model {actualWorld = actualWorld m, epistemicState = Set.filter
    filterfunc  (epistemicState m), domain = domain m} where
    filterfunc = (\w -> satisfiesModel (Model {actualWorld = w, epistemicState =
        epistemicState m, domain = domain m}) alpha)
```

In addition to ASK and $TELL$, it is valuable to define an inital epistemic state. $INITIAL$ is the epistemic state before any $TELL$ operations. This state contains all possible world states as there is nothing known that eliminates any possible world.

```
-- INITIAL operation (Section 5.3)
-- Generate all possible world states for a finite set of atoms and terms
allWorldStates :: [PrimitiveAtom] -> [PrimitiveTerm] -> [StdName] -> [WorldState]
allWorldStates atoms terms dom = do
   atomVals <- mapM (\_ -> [True, False]) atoms
   termVals <- mapM (\_ -> dom) terms
   return $ mkWorldState (zip atoms atomVals) (zip terms termVals)

initial :: [PrimitiveAtom] -> [PrimitiveTerm] -> [StdName] -> EpistemicState
initial atoms terms dom = Set.fromList (allWorldStates atoms terms dom)
```

3 Tableau-Based Satisfiability and Validity Checking in $\mathcal{KL}$

Note: For the Beta-version, we omitted function symbol evaluation, limiting the satisfiability and validity checking to a propositional-like subset.
This subsection implements satisfiability and validity checkers for $\mathcal{KL}$ using the tableau method, a systematic proof technique that constructs a tree to test formula satisfiability by decomposing logical components and exploring possible models. In $\mathcal{KL}$, this requires handling both first-order logic constructs (quantifiers, predicates) and the epistemic operator K, which requires tracking possible worlds. Note that the full first-order epistemic logic with infinite domains is in general undecidable (Levesque and Lakemeyer 2001 p. 173), so we adopt a semi-decision procedure: it terminates with "satisfiable" if an open branch is found but may loop infinitely for unsatisfiable cases due to the infinite domain $\mathcal{N}$. The Tableau module builds on SyntaxKL and SemanticsKL:

```
module Tableau where

import SyntaxKL
import SemanticsKL
import Data.Set (Set)
import qualified Data.Set as Set
```

Tableau Approach
The tableau method tests satisfiability as follows: A formula α is satisfiable if there exists an epistemic state e and a world $w \in e$ such that $e, w \models \alpha$. The tableau starts with α and expands it, seeking an open (non-contradictory) branch representing a model. A formula α is valid if it holds in all possible models ($e, w \models \alpha$ for all e, w). We test validity by checking if $\neg\alpha$ unsatisfiable (i.e., all tableau branches close). For $\mathcal{KL}$ we have to handle two things:

- Infinite domains: $\mathcal{KL}$ assumes a countably infinite set of standard names (Levesque and Lakemeyer 2001, p.23). The tableau method handles this via parameters (free variables) and δ-rules (existential instantiation), introducing new names as needed. This means that we Use a countably infinite supply of parameters (e.g., a1, a2,...) instead of enumerating all standard names.
- Modal handling: The K-operator requires branching over possible worlds within an epistemic state.

First, we define new types for the tableau node and branch: Nodes pair formulas with world identifiers, and branches track nodes and used parameters.

```
-- A tableau node: formula labeled with a world
data Node = Node Formula World deriving (Eq, Show)

type World = Int    -- World identifier (0, 1, ...)
```

```
-- A tableau branch: list of nodes and set of used parameters
data Branch = Branch { nodes :: [Node], params :: Set StdName } deriving (Show)
```

Tableau Rules

Rules decompose formulas, producing either a closed branch (contradictory) or open branches (consistent). applyRule implements these rules, handling logical and epistemic operators. The rules are applied iteratively to unexpanded nodes until all branches are either closed or fully expanded (open).

```
-- Result of applying a tableau rule
data RuleResult = Closed | Open [Branch] deriving (Show)

-- Generates fresh parameters not in the used set
newParams :: Set StdName -> [StdName]
newParams used = [StdName ("a" ++ show i) | i <- [(1::Int)..], StdName ("a" ++ show i) `Set
    .notMember` used]

-- Applies tableau rules to a node on a branch
applyRule :: Node -> Branch -> RuleResult
applyRule (Node f w) branch = case f of
  Atom _ -> Open [branch]    -- If formula is an atom: Do nothing; keep the formula in the
      branch.
  Not (Atom _) -> Open [branch]   -- Negated atoms remain, checked by isClosed
  Equal _ _ -> Open [branch]   -- Keep equality as is; closure checks congruence
  Not (Equal _ _) -> Open [branch]   -- Keep negated equality
  Not (Not f') -> Open [Branch (Node f' w : nodes branch) (params branch)] -- Case: double
      negation, e.g., replace $\neg \neg \varphi$ with $\varphi$
  Not (Or f1 f2) -> Open [Branch (Node (Not f1) w : Node (Not f2) w : nodes branch) (params
       branch)] -- Case: negated disjunction
  Not (Exists x f') -> Open [Branch (Node (klforall x (Not f')) w : nodes branch) (params
      branch)] -- Case:: negated existential
  Not (K f') -> Open [expandKNot f' w branch] -- Case: negated knowledge
  Or f1 f2 -> Open [ Branch (Node f1 w : nodes branch) (params branch)
                   , Branch (Node f2 w : nodes branch) (params branch) ] -- Disjunction
                       rule, split the branch
  Exists x f' ->    -- Existential rule ($\delta$-rule), introduce a fresh parameter a (e.g
      .,  a 1  ) not used elsewhere, substitute x with a, and continue
    let newParam = head (newParams (params branch))
        newBranch = Branch (Node (subst x newParam f') w : nodes branch)
                           (Set.insert newParam (params branch))
    in Open [newBranch]
  K f' -> Open [expandK f' w branch] -- Knowledge rule, add formula to a new world

-- Expands formula K \varphi to a new world
expandK :: Formula -> World -> Branch -> Branch
expandK f w branch = Branch (Node f (w + 1) : nodes branch) (params branch)

-- Expands \not K \varphi to a new world
expandKNot :: Formula -> World -> Branch -> Branch
expandKNot f w branch = Branch (Node (Not f) (w + 1) : nodes branch) (params branch)
```

Branch Closure

isClosed determines whether a tableau branch is contradictory (closed) or consistent (open). A branch closes if it contains an explicit contradiction, meaning no model can satisfy all the formulas in that branch. If a branch is not closed, it is potentially part of a satisfiable interpretation. The input is a Branch, which has nodes :: [Node] (each Node f w is a formula f in world w) and params :: Set StdName (used parameters). The function works as follows: first, we collect the atoms ((a, w, True) for positive atoms (Node (Atom a) w); (a, w, False) for negated Atoms (Node (Not (Atom a)) w)). For example, if nodes = [Node (Atom P(n1)) 0, Node (Not (Atom P(n1))) 0], then atoms = [(P(n1), 0, True), (P(n1), 0, False)]. Next, we collect the equalities. After this, we check the atom contradictions. There we use *any* to find pairs in *atoms* and return True if a contradiction exists. In a subsequent step, we check for equality contradictions. The result of the function is atomContra || eqContra: this is True if either type of contradiction is found and False otherwise. This function reflects the semantic requirement that a world state w in an epistemic state e can not assign both True and False to the same ground atom or equality

```
-- Branch closure with function symbols
isClosed :: Branch -> Bool
isClosed b =
  let atoms = [(a, w, True) | Node (Atom a) w <- nodes b]
              ++ [(a, w, False) | Node (Not (Atom a)) w <- nodes b]
      equals = [((t1, t2), w, True) | Node (Equal t1 t2) w <- nodes b]
               ++ [((t1, t2), w, False) | Node (Not (Equal t1 t2)) w <- nodes b]
      atomContra = any (\(a1, w1, b1) -> any (\(a2, w2, b2) -> a1 == a2 && w1 == w2 && b1
          /= b2) atoms) atoms
      eqContra = any (\((t1, t2), w1, b1) -> any (\((t3, t4), w2, b2) ->
                   t1 == t3 && t2 == t4 && w1 == w2 && b1 /= b2) equals) equals
  in atomContra || eqContra  -- True if any contradiction exists
```

Tableau Expasion

Next, we have the function expandTableau. expandTableau iteratively applies tableau rules to expand all branches, determining if any remain open (indicating satisfiability). It returns Just branches if at least one branch is fully expanded and open, and Nothing if all branches close. This function uses recursion. It continues until either all branches are closed or some are fully expanded

```
-- Expands the tableau, returning open branches if satisfiable
expandTableau :: [Branch] -> Maybe [Branch]
expandTableau branches
  | all isClosed branches = Nothing --If every branch is contradictory, return Nothing
  | any (null . nodes) branches = Just branches --If any branch has no nodes left to expand
      (and isn't closed), it's open and complete
  | otherwise = do
      let (toExpand, rest) = splitAt 1 branches --Take the first branch (toExpand) and
          leave the rest.
          branch = head toExpand --Focus on this branch.
          node = head (nodes branch) --Pick the first unexpanded node.
          remaining = Branch (tail (nodes branch)) (params branch) --he branch minus the
              node being expanded.
      case applyRule node remaining of
        Closed -> expandTableau rest --Skip this branch, recurse on rest.
        Open newBranches -> expandTableau (newBranches ++ rest) --Add the new branches (e.g
            ., from \lor or \exists) to rest, recurse.
```

Top-Level Checkers

As top-level function we use isSatisfiable and isValid. isSatisfiable tests whether a formula f has a satisfying model. It starts the tableau process and interprets the result. This function gets a Formula f as an input and then creates a single branch with Node f 0 (formula f in world 0) and an empty set of parameters. Next, it calls expandTablaeu on this initial branch. It then interprets the result: if expandTableau returns Just _, this means, that at least one open branch exists, thus, the formula is satisfiable. If expandTableau returns Nothing, this means that all branches are closed and the formula is unsatisfiable.

```
-- Tests if a formula is satisfiable
isSatisfiable :: Formula -> Bool
isSatisfiable f = case expandTableau [Branch [Node f 0] Set.empty] of
  Just _ -> True
  Nothing -> False
```

The three function isSatisfiable, expandTableau, and isClosed interact as follows: isSatisfiable starts the process with a single branch containing the formula. expandTableau recursively applies applyRule to decompose formulas, creating new branches as needed (e.g., for $\lor$, $\exists$).

isClosed checks each branch for contradictions, guiding expandTableau to prune closed branches or halt with an open one.

```
-- Tests if a formula is valid
isValid :: Formula -> Bool
isValid f = not (isSatisfiable (Not f))
```

4 Semantics Tests

```
{-# LANGUAGE InstanceSigs #-}

module Main where

import SemanticsKL
import SyntaxKL
import Generators

import qualified Data.Map as Map
import qualified Data.Set as Set

import Test.Hspec
import Test.QuickCheck
import Control.Exception (evaluate)
```

The following tests are for the semantics of $\mathcal{KL}$, which are defined in the SemanticsKL module. The tests are written using the Hspec testing framework and QuickCheck for property-based testing. The tests cover the evaluation of terms, formulas, and models, as well as model checking function. The Generators file provides helper functions for generating implementing testing, but have been omitted for brevity.

```
main :: IO ()
main = hspec $ do
    describe "Eq is derived for the relevant KL Semantics" $ do
        it "WorldState Eq is derived" $ do
            property $ \w -> w == (w :: WorldState)
        -- TODO: add test to be sure that the world state only has primitive terms and
            atoms
    describe "evalTerm" $ do
        it "evalTerm errors with variables" $ do
            property $ \x -> evaluate (evalTerm (WorldState Map.empty Map.empty) (VarTerm x
                )) `shouldThrow` anyException
        it "evalTerm returns the StdName for StdNameTerm" $ do
            property $ \w n -> evalTerm w (StdNameTerm n) == n
        it "evalTerm returns the StdName after applying all functions (depth 2)" $ do
            let n1 = StdName "n1"
                n2 = StdName "n2"
                n3 = StdName "n3"
                n4 = StdName "n4"
                w = WorldState Map.empty (Map.fromList [
                                            (FuncAppTerm "f" [StdNameTerm n1, StdNameTerm
                                                n2], n3),
                                            (FuncAppTerm "g" [StdNameTerm n4], n1)
                                        ])
                t = FuncAppTerm "f" [FuncAppTerm "g" [StdNameTerm n4], StdNameTerm n2]
            evalTerm w t `shouldBe` StdName "n3"
    describe "isGround" $ do
        it "isGround returns False for VarTerm" $ do
            property $ \n -> not $ isGround (VarTerm n)
        it "isGround returns True for GroundFuncAppTerm or GroundStdNameTerm" $ do
          property $ forAll genGroundTerm $ \t -> isGround (t :: Term)
        it "isGround returns False for complex FuncAppTerm with a non-ground argument" $ do
            let term = FuncAppTerm "f" [FuncAppTerm "g" [VarTerm $ Var "x", StdNameTerm $
                StdName "n1"]]
            isGround term `shouldBe` False
    describe "isGroundFormula" $ do
        it "isGroundFormula returns True for groundFormula" $ do
```

```
            property $ forAll genGroundFormula $ \f -> isGroundFormula (f :: Formula)
        it "isGroundFormula returns False for Exists" $ do
            property $ \n f -> not $ isGroundFormula (Exists (Var n) (f :: Formula))
        it "isGroundFormula returns False for Atom with a non-ground term" $ do
            isGroundFormula (Atom (Pred "P" [VarTerm $ Var "x"])) `shouldBe` False
        it "isGroundFormula returns False for Equal with a non-ground term" $ do
            isGroundFormula (Equal (VarTerm $ Var"x") (StdNameTerm $ StdName "n1")) `
                shouldBe` False
    describe "substTerm" $ do
        it "substTerm replaces the variable with the StdName" $ do
            let term = FuncAppTerm "f" [VarTerm $ Var "x", StdNameTerm $ StdName "n1"]
            substTerm (Var "x") (StdName "n2") term `shouldBe` FuncAppTerm "f" [StdNameTerm
                 $ StdName "n2", StdNameTerm $ StdName "n1"]
        it "substTerm does not replace the wrong variable" $ do
            let term = FuncAppTerm "f" [VarTerm $ Var "y", StdNameTerm $ StdName "n1"]
            substTerm (Var "x") (StdName "n2") term `shouldBe` term
    describe "subst" $ do
        it "subst replaces the variable with the StdName in an Atom" $ do
            let atom = Atom (Pred "P" [VarTerm $ Var "x"])
            show (subst (Var "x") (StdName "n1") atom) `shouldBe` show (Atom (Pred "P" [
                StdNameTerm $ StdName "n1"]))
        it "subst replaces the variable with the StdName in an Equal" $ do
            let formula = Equal (VarTerm $ Var "x") (StdNameTerm $ StdName "n1")
            show (subst (Var "x") (StdName "n2") formula) `shouldBe` show (Equal (
                StdNameTerm $ StdName "n2") (StdNameTerm $ StdName "n1"))
        it "subst replaces the variable with the StdName in a Not formula" $ do
            let formula = Not (Atom (Pred "P" [VarTerm $ Var "x"]))
            show (subst (Var "x") (StdName "n1") formula) `shouldBe` show (Not (Atom (Pred
                "P" [StdNameTerm $ StdName "n1"])))
        it "subst replaces the variable with the StdName in an Or formula" $ do
            let formula = Or (Atom (Pred "P" [VarTerm $ Var "x"])) (Atom (Pred "Q" [VarTerm
                 $ Var "y"]))
            show (subst (Var "x") (StdName "n1") formula) `shouldBe` show (Or (Atom (Pred "
                P" [StdNameTerm $ StdName "n1"])) (Atom (Pred "Q" [VarTerm $ Var "y"])))
        it "subst replaces the variable with the StdName in an Exists if the variable not
            in Exists scope" $ do
            let formula = Exists (Var "x") (Atom (Pred "P" [VarTerm $ Var "x", VarTerm $
                Var "y"]))
            -- replaces y with n2
            show (subst (Var "y") (StdName "n2") formula) `shouldBe` show (Exists (Var "x")
                 (Atom (Pred "P" [VarTerm $ Var "x", StdNameTerm $ StdName "n2"])))
        it "subst does not replace the variable with the StdName in Exists if the variable
            is in the Exists scope" $ do
            let formula = Exists (Var "x") (Atom (Pred "P" [VarTerm $ Var "x", VarTerm $
                Var "y"]))
            -- does not replace x with n2
            show (subst (Var "x") (StdName "n2") formula) `shouldBe` show formula
        it "subst replaces the variable with the StdName in a K formula" $ do
            let formula = K (Atom (Pred "P" [VarTerm $ Var "x"]))
            show (subst (Var "x") (StdName "n2") formula) `shouldBe` show (K (Atom (Pred "P
                " [StdNameTerm $ StdName "n2"])))
    describe "satisfiesModel" $ do
        let x = Var "x"
            n1 = StdNameTerm $ StdName "n1"
            n2 = StdNameTerm $ StdName "n2"
            p = Atom (Pred "P" [])
            px = Atom (Pred "P" [VarTerm x])
            py = Atom (Pred "P" [VarTerm $ Var "y"])
            pt = Atom (Pred "P" [n1])
        context "satisfiesModel satisfies validities when atoms are ground" $ do
            it "satisfiesModel satisfies P -> ~~ P" $ do
                property $ \m -> satisfiesModel m (Or (Not p) (Not (Not p))) `shouldBe`
                    True
            it "satisfiesModel satisfies P(t) -> ~~ P(t)" $ do
                property $ \m -> satisfiesModel m (Or (Not pt) (Not (Not pt))) `shouldBe`
                    True
            it "satisfiesModel errors for P(x) -> ~~ P(x)" $ do
                property $ \m -> evaluate (satisfiesModel m (Or (Not px) (Not (Not px)))) `
                    shouldThrow` anyException
            it "satisfiesModel satisfies t=t" $ do
                property $ \m -> satisfiesModel m (Equal n1 n1) `shouldBe` True
            it "satisfiesModel errors for x=x" $ do
                property $ \m -> evaluate (satisfiesModel m (Equal (VarTerm x) (VarTerm x))
```

```
                ) `shouldThrow` anyException
        it "satisfiesModel satisfies ForAll x (P(x) -> P(x))" $ do
            property $ \m -> satisfiesModel m (Not (Exists x (Not (Or (Not px) px)))) `
                shouldBe` True
        it "satisfiesModel satisfies ForALL x (P(x) -> ~~ P(x))" $ do
            property $ \m -> satisfiesModel m (Not (Exists x (Not (Or (Not px) (Not (
                Not px))))) ) `shouldBe` True
        it "satisfiesModel satisfies ForAll x (P(x) -> Exists y P(y))" $ do
            property $ \m -> satisfiesModel m (Not (Exists x (Not (Or (Not px) (Exists
                (Var "y") py)) ))) `shouldBe` True
        it "satisfiesModel satisfies ((n1 = n2) -> K (n1 = n2)" $ do
            property $ \m -> satisfiesModel m (Or (Not (Equal n1 n2)) (K (Equal n1 n2))
                ) `shouldBe` True
        it "satisfiesModel satisfies ((n1 /= n2) -> K (n1 /= n2)" $ do
            property $ \m -> satisfiesModel m (Or (Not (Not (Equal n1 n2))) (K (Not (
                Equal n1 n2)))) `shouldBe` True
        it "satisfiesModel satisfies (K alpha -> K K alpha)" $ do
            property $ \m -> satisfiesModel m (Or (Not (K pt)) (K (K pt))) `shouldBe`
                True
        it "satisfiesModel satisfies (~K alpha -> K ~K alpha)" $ do
            property $ \m -> satisfiesModel m (Or (Not (Not (K pt))) (K (Not (K pt))))
                `shouldBe` True
    context "satisfiesModel does not satisfy contradictions when atoms are ground" $ do
        it "satisfiesModel does not satisfy ~(P v ~P)" $ do
            property $ \m -> satisfiesModel m (Not (Or p (Not p))) `shouldBe` False
        it "satisfiesModel does not satisfy (Exists x (x /= x))" $ do
            property $ \m -> satisfiesModel m (Exists x (Not (Equal (VarTerm x) (
                VarTerm x)))) `shouldBe` False
describe "freeVars" $ do
        let x = Var "x"
            y = Var "y"
            n1 = StdNameTerm $ StdName "n1"
            n2 = StdNameTerm $ StdName "n2"
            px = Atom (Pred "P" [VarTerm x])
            py = Atom (Pred "P" [VarTerm y])
            pf = Atom (Pred "P" [FuncAppTerm "f" [VarTerm x], FuncAppTerm "g" [VarTerm
                y]])
        it "freeVars returns nothing if no free var in formula" $ do
            let f = Exists x (Or (Or (Not px) (Exists y py)) (Equal n1 n2))
            freeVars f `shouldBe` Set.fromList []
        it "freeVars returns the free variables in a simple formula" $ do
            let f = Or (Or (Not px) py) (Equal n1 n2)
            freeVars f `shouldBe` Set.fromList [x, y]
        it "freeVars returns the free variables in a complex formula" $ do
            let f = Exists x (Or (Or (Not px) pf) (Equal n1 n2))
            freeVars f `shouldBe` Set.fromList [y]
describe "groundFormula" $ do
    it "groundFormula returns a ground formula (dependant on isGroundFormula passing
        all tests)" $ do
        property $ forAll (resize 5 arbitrary) $ \f ->
            forAll genStdNameSet $ \s ->
                all isGroundFormula (groundFormula (f :: Formula) s)
describe "checkModel" $ do
    let x = Var "x"
        px = Atom (Pred "P" [VarTerm x])
    context "checkModel satisfies validities when atoms are unground" $ do
        it "checkModel errors for P(x) -> ~~ P(x)" $ do
            property $ \m -> checkModel m (Or (Not px) (Not (Not px))) `shouldBe` True
        it "checkModel errors for x=x" $ do
            property $ \m -> checkModel m (Equal (VarTerm x) (VarTerm x)) `shouldBe`
                True
```

5 How to Use the Code

In this section we will provide instructions and examples on how to use our code.

5.1 Syntax and Semantics

To use the semantic evaluation functions from the SemanticsKL module in GHCi, begin by saving the file (e.g., SemanticsKL.lhs) in your working directory. Start GHCi by typing ghci in your terminal from that directory, then load the module with :load SemanticsKL.lhs, ensuring SyntaxKL.lhs is also present and correctly imported. After loading, you can interactively test $\mathcal{KL}$ models and formulas. For instance, create a simple world state with let ws = mkWorldState [(PPred "P" [StdName "n1"], True)] [], a model with let m = Model ws (Set.singleton ws) (Set.fromList [StdName "n1"]), and a formula like let f = Atom (Pred "P" [StdNameTerm (StdName "n1")]). Check if the formula holds using checkModel m f, which should return True since $P(n1)$ is true in the model. Alternatively, evaluate satisfiability with satisfiesModel m f.

5.2 Satisfiability and Validity Checking

To use the tableau-based satisfiability and validity checkers from the Tableau module in GHCi, first ensure the file (e.g., Tableau.lhs) is saved in your working directory. Launch GHCi from that directory by running ghci in your terminal. Load the module with :load Tableau.lhs, which will compile and make its functions available, assuming SyntaxKL.lhs and SemanticsKL.lhs are also in the same directory and properly imported. Once loaded, you can test formulas interactively. For example, define a formula like let f = Or (Atom (Pred "P" [StdNameTerm (StdName "n1")])) (Not (Atom (Pred "P" [StdNameTerm (StdName "n1")]))) and check its satisfiability with isSatisfiable f, which should return True (since $P(n1) \vee \neg P(n1)$ is satisfiable). Similarly, test validity with isValid f, which returns False. Use :reload to update changes after editing the file, and :quit to exit GHCi.

5.3 Ask and Tell

To use the AskTell module's ASK, TELL, and INITIAL operators in GHCi, set up the file as described in the previous subsections. Once loaded, you can experiment with epistemic operations. For example, define a domain with let d = Set.fromList [StdName "n1"], an initial epistemic state with let e = initial [PPred "P" [StdName "n1"]] [] [StdName "n1"], and a formula like let f = Atom (Pred "P" [StdNameTerm (StdName "n1")]). Test the ASK operator with ask d e f, which checks if $P(n1)$ is known across all worlds. Apply TELL with let e' = tell d e f to filter worlds where $P(n1)$ holds, then verify with ask d e' f, expecting True. Alternatively, use askModel and tellModel with a Model, such as let m = Model (Set.findMin e) e d, via askModel m f and tellModel m f.

5.4 Translations

You can interactively test translations and model properties. To do so, you can, for example, define a KL formula like let f = K (Atom (Pred "P" [StdNameTerm (StdName "n1")])) and translate it to standard epistemic logic (SEL) with translateFormToKr f, expecting Just (Box (P 1)). Conversely, translate an SEL formula like let g = Box (P 1) to KL with translateFormToKL g, yielding K (Atom (Pred "P" [StdNameTerm (StdName "n1")])). For models, create a KL model with let m = model1 and convert it to a Kripke model using translateModToKr m, then compare with kripkeM1 via test1. You can try out the Kripke-to-KL translation by using kripkeToKL exampleModel3 (makeWorldState 30), checking if it succeeds. You could also run some of our predefined tests like ftest1 or testAllFormulae to get to know our translation functions.

5.5 Tests

You can run all the tests and examine the current code coverage run ‘stack clean stack test –coverage‘.’

References

Levesque, Hector J (1981). “The Interaction with Incomplete Knowledge Bases: A Formal Treatment.” In: *IJCAI*, pp. 240–245.

Levesque, Hector J and Gerhard Lakemeyer (2001). *The logic of knowledge bases*. Mit Press.