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

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

Saturday 29th 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 was first developed by Levesque (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 language $\mathcal{KL}$ is built on the following alphabet:

- **Variables**: $x, y, z, \dots$ (an infinite set).
- **Constants**: $c, d, n1, n2, \dots$ (including standard names).
- **Function symbols**: $f, g, h, \dots$ (with associated arities).
- **Predicate symbols**: $P, Q, R, \dots$ (with associated arities).
- **Logical symbols**: $\neg, \vee, \exists, =$,K, (,).

In this our 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.

```
--TODO hide
arbitraryUpperLetter :: Gen String
arbitraryUpperLetter = (:[]) <$> elements ['A'..'Z']
--TODO hide
arbitraryLowerLetter :: Gen String
arbitraryLowerLetter = (:[]) <$> elements ['a'..'z']

-- Represents a standard name (e.g., "n1") from the infinite domain N
newtype StdName = StdName String deriving (Eq, Ord, Show)

--TODO hide
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)

--TODO hide
instance Arbitrary Variable where
  arbitrary:: Gen Variable
  arbitrary = Var . show <$> elements [1 .. 20::Int]
```

Terms, Atoms, and Formulas
The syntax of $\mathcal{KL}$ is defined recursively in Backus-Naur Form as follows:
Terms represent objects in the domain:

```
<term> ::= <variable> | <constant> | <function-term>
<variable> ::= "x" | "y" | "z" | ...
<constant> ::= "c" | "d" | "n1" | "n2" | ...
```

```
<function-term> ::= <function-symbol> "(" <term-list> ")"
<function-symbol> ::= "f" | "g" | "h" | ...
<term-list> ::= <term> | <term> "," <term-list>
```

Well-formed formulas (wffs) define the logical expressions:

```
<wff> ::= <atomic-wff> | <negated-wff> | <disjunction-wff> |
          <existential-wff> | <knowledge-wff>
<atomic-wff> ::= <predicate> | <equality>
<predicate> ::= <predicate-symbol> "(" <term-list> ")"
<predicate-symbol> ::= "P" | "Q" | "R" | ...
<equality> ::= <term> "=" <term>
<negated-wff> ::= "\not" <wff>
<disjunction-wff> ::= "(" <wff> "\lor" <wff> ")"
<existential-wff> ::= "\exists" <variable> "." <wff>
<knowledge-wff> ::= "K" <wff>
```

Predicate and function symbols have implicit arities, abstracted here for generality. Furthermore, the epistemic operator $\mathbf{K}$ allows nested expressions, e.g., $\mathbf{K}\neg\mathbf{K}P(x)$.
Sentences of $\mathcal{KL}$ can look like this:

- $Teach(ted, sue)$:
  ```
  <wff> -> <atomic-wff> -> <predicate> -> "Teach" "(" "ted" "," "sue" ")"
  ```
- $\boldsymbol{K}\exists x.Teach(x, sam)$:
  ```
  <wff> -> <knowledge-wff> -> "K" <wff>
        -> "K" <existential-wff>  "K" "\exists" "x" "." <wff>
        -> "K" "\exists" "x" "." "Teach" "(" "x" "," "sam" ")"
  ```
- $\neg\boldsymbol{K}Teach(tina, sue)$:
  ```
  <wff> -> <negated-wff> -> "\neg" <wff>
        -> "\neg" <knowledge-wff> -> "\neg" "K" <wff>
        -> "\neg" "K" "Teach" "(" "tina" "," "sue" ")"
  ```

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`.

```haskell
-- 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)

--TODO hide
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)

--TODO hide
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)

--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)

--TODO hide
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 $\mathcal{KL}$

As we have seen in the previous section, $\mathcal{KL}$ is an epistemic extension of first-order logic. The main differences to classical first-order logic are that introduces a knowledge operator **K** and uses an infinite domain $\mathcal{N}$ of standard names to denote individuals. It is designed to model knowledge and uncertainty, as detailed in Levesque and Lakemeyer (2001).
Formulas of $\mathcal{KL}$ 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. We have implemented it as mapping to atoms and terms instead of just primitive ones, as we make sure when creating a `WorldState` to only use primitive atoms and primitive terms (by the function `mkWorldState`). An `EpistemicState`, defined as a set of `WorldState`s, 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)

--TODO: hide
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 `Atom` and `Term` (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). The function `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
      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))
```

```haskell
-- 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`:

- `VarTerm _`: Errors, as only ground terms (no free variables) are valid (Levesque and Lakemeyer 2001, p. 24).
- `StdNameTerm n`: Returns `n`, since standard names denote themselves (ibid., p. 22).
- `FuncAppTerm f args`: Recursively evaluates `args` to `StdName`s, builds a ground `FuncAppTerm`, and looks up its value in `termValues w`, erroring if undefined.

```haskell
-- 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 a capture. We need these functions to be able to define a function that checks whether a formula is true in a `WorldState` and `EpistemicState`.

```haskell
-- 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)
```

Truth in a Model

Since we want to be able check if a formula is true in a model, we want to make the model explicit:

```haskell
-- 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, Eq)

--TODO hide
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 **K**-operator. The function `satisfiesModel` implements $\mathcal{KL}$'s satisfaction relation, checking truth across worlds.

```haskell
-- Checks if a formula is true in a model
satisfiesModel :: Model -> Formula -> Bool
satisfiesModel (Model w _ _) (Atom (Pred p terms)) =
  if all isGround terms
    then Map.findWithDefault False (Pred p terms) (atomValues w)
    else error "Non-ground atom in satisfiesModel!"
satisfiesModel (Model w _ _) (Equal t1 t2) =
  if isGround t1 && isGround t2
    then evalTerm w t1 == evalTerm w t2
    else error "Non-ground equality in satisfiesModel!"
satisfiesModel (Model w e d) (Not f) = not (satisfiesModel (Model w e d) f)
```

```
satisfiesModel (Model w e d) (Or f1 f2) = satisfiesModel (Model w e d) f1 || satisfiesModel
     (Model w e d) f2
satisfiesModel (Model w e d) (Exists x f) = any (\n -> satisfiesModel (Model w e d) (subst
    x n f)) (Set.toList d)
satisfiesModel (Model _ e d) (K f) = all (\w' -> satisfiesModel (Model w' e d) 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 throw an error or always substitute the same standard name. The implementation that we have chosen is more flexible and allows for more varied usage, however it is computationally expensive. We still decided to handle free variables in this way, as this implementation is the most faithful to the theory as described in Levesque and Lakemeyer (2001). 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
                     else Set.delete x (vars f') -- Exclude bound variable x
      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 known sentences there are, the smaller 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 SyntaxKL
import SemanticsKL
import qualified Data.Set as Set
```

The *ask* operator determines whether or not a formula is known to a knowledge base. Formally, given an epistemic state e and any sentence α of $\mathcal{KL}$,

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

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 a formula 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
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 initial 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
-- 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
    | null atoms && null terms = Set.empty
    | otherwise = Set.fromList (allWorldStates atoms terms dom)
```

3 Comparing KL and Epistemic Logic

```
module Translator where

import Data.List
import Data.Maybe
import GHC.Num
import Test.QuickCheck
import qualified Data.Map as Map
import qualified Data.Set as Set
import Control.Monad (replicateM)
```

```haskell
-- importing syntax and semantics of KL
import SyntaxKL
import SemanticsKL
```

We want to compare $\mathcal{KL}$ and Propositional Modal Logic based on Kripke frames. (Call this PML). For example, we might want to compare the complexity of model checking for $\mathcal{KL}$ and PML. To do this, we need some way of "translating" between formulas of $\mathcal{KL}$ and formulas of PML, and between $\mathcal{KL}$-models and Kripke models. This would allow us to, e.g., (1) take a set of $\mathcal{KL}$-formulas of various lengths and a set of $\mathcal{KL}$-models of various sizes; (2) translate both formulas and models into PML; (3) do model checking for both (i.e.. on the $\mathcal{KL}$ side, and on the PML side); (4) compare how time and memory scale with length of formula.
Three things need to be borne in mind when designing the translation functions:

1. The language of $\mathcal{KL}$ is predicate logic, plus a knowledge operator **K**. The language of PML, on the other hand, is propositional logic, plus a knowledge operator.
2. Kripke models are much more general than $\mathcal{KL}$ models.
3. In Kripke models, there is such a thing as evaluating a formula at various different worlds, whereas this has no equivalent in $\mathcal{KL}$-models.

We deal with the first two points by making some of the translation functions partial; we deal with the third, by, in effect, translating $\mathcal{KL}$ models to pointed Kripke models. Details will be explained in the sections on the respective translation functions below.

3.1 Syntax and Semantics of PML

The syntax and semantics of PML is well-known: the language is just the language of basic modal logic, where the Box operator $\Box$ is interpreted as "It is known that...". Models are Kripke models. A mathematical description of all this can be found in any standard textbook on modal logic, so we focus on the implementation, here.

Syntax
The implementation of PML syntax in Haskell is straightforward.

```haskell
data ModForm = P Proposition
             | Neg ModForm
             | Dis ModForm ModForm
             | Box ModForm
             deriving (Eq,Ord,Show)

dia :: ModForm -> ModForm
dia f = Neg (Box (Neg f))

con :: ModForm -> ModForm -> ModForm
con f g = Neg (Dis (Neg f) (Neg g))

impl :: ModForm -> ModForm -> ModForm
impl f = Dis (Neg f)

-- this will be useful for testing later
instance Arbitrary ModForm where
  arbitrary = resize 16 (sized randomForm) where
    randomForm :: Int -> Gen ModForm
    randomForm 0 = P <$> elements [1..5]
```

```haskell
    randomForm n = oneof [ Neg <$> randomForm (n `div` 2)
                         , Dis <$> randomForm (n `div` 2)
                               <*> randomForm (n `div` 2)
                         , Box <$> randomForm (n `div` 2) ]
```

Semantics

For some parts of our project, it will be most convenient to let Kripke models have `WorldStates` (as defined in `SemanticsKL`) as worlds; for others, to have the worlds be `Integers`. We therefore implement Kripke models as a polymorphic data type, as follows:

```haskell
--definition of models
type World a = a
type Universe a = [World a]
type Proposition = Int

type Valuation a = World a -> [Proposition]
type Relation a = [(World a,World a)]

data KripkeModel a = KrM
   { universe :: Universe a
   , valuation :: Valuation a
   , relation :: Relation a}

--definition of truth for modal formulas
--truth at a world
makesTrue :: Eq a => (KripkeModel a, World a) -> ModForm -> Bool
makesTrue (KrM _ v _, w) (P k)     = k `elem` v w
makesTrue (m,w)          (Neg f)   = not (makesTrue (m,w) f)
makesTrue (m,w)          (Dis f g) = makesTrue (m,w) f || makesTrue (m,w) g
makesTrue (KrM u v r, w) (Box f)   = all (\w' -> makesTrue (KrM u v r,w') f) (r ! w)

(!) :: Eq a => Relation a -> World a -> [World a]
(!) r w = map snd $ filter ((==) w . fst) r

--truth in a model
trueEverywhere :: Eq a => KripkeModel a -> ModForm -> Bool
trueEverywhere (KrM x y z) f = all (\w -> makesTrue (KrM x y z, w) f) x
```

We will also have to be able to check whether a formula is valid on the frame underlying a Kripke model. This is implemented as follows:

```haskell
-- Maps Propositional Modal Logic to a KL atom
propToAtom :: Proposition -> Atom
propToAtom n = Pred "P" [StdNameTerm (StdName ("n" ++ show n))]  -- e.g., 1 -> P(n1)

-- Creates a KL WorldState from a list of propositional variables????
createWorldState :: [Proposition] -> WorldState
createWorldState props =
  let atomVals = Map.fromList [(propToAtom p, True) | p <- props]  -- Maps each proposition
       to True
      termVals = Map.empty                                          -- No term valuations
          needed here
  in WorldState atomVals termVals

-- extract all the propositional variables of a Propositional Modal Logic formula
uniqueProps :: ModForm -> [Proposition]
uniqueProps f = nub (propsIn f)
  where
    propsIn (P k)       = [k]
    propsIn (Neg g)     = propsIn g
    propsIn (Dis g h)   = propsIn g ++ propsIn h
    propsIn (Box g)     = propsIn g

-- Generate all possible valuations explicitly
allValuations :: Ord a => [World a] -> [Proposition] -> [Valuation a]
allValuations univ props =
  let subsetsP = subsequences props
      allAssignments = replicateM (length univ) subsetsP
```

```haskell
  in [ \w -> Map.findWithDefault [] w (Map.fromList (zip univ assignment))
     | assignment <- allAssignments ]

-- Checks whether a Kripke formula is valid on a given Kripke model
isValidKr :: (Eq a, Ord a) => ModForm -> KripkeModel a -> Bool
isValidKr f (KrM univ _ rel) =
  let props = uniqueProps f
      valuations = allValuations univ props
  in all (\v -> all (\w -> makesTrue (KrM univ v rel, w) f) univ) valuations
```

Sometimes it will be useful to convert between models of type `KripkeModel WorldState` and models of type `KripkeModel Integer`. To enable this, we provide the following functions:

```haskell
translateKrToKrInt :: KripkeModel WorldState -> KripkeModel Integer
translateKrToKrInt (KrM u v r) = KrM u' v' r' where
   ur = nub u -- the function first gets rid of duplicate worlds in the model
   u' = take (length ur) [0..]
   v' n = v (intToWorldState ur n) where
      intToWorldState :: Universe WorldState -> Integer -> WorldState
      intToWorldState urc nq = urc !! integerToInt nq
   r' = [(worldStateToInt ur w, worldStateToInt ur w') | (w,w') <- r] where
      worldStateToInt :: Universe WorldState-> WorldState -> Integer
      worldStateToInt uni w = toInteger $ fromJust $ elemIndex w uni

convertToWorldStateModel :: KripkeModel Integer -> KripkeModel WorldState
convertToWorldStateModel (KrM intUniv intVal intRel) =
  let worldStates = map makeWorldState intUniv
      worldToInt :: WorldState -> Integer
      worldToInt ws = case find (\(_, w) -> w == ws) (zip intUniv worldStates) of
                        Just (i, _) -> i
                        Nothing -> error "WorldState not found in universe"
      newVal :: Valuation WorldState
      newVal ws = intVal (worldToInt ws)
      newRel :: Relation WorldState
      newRel = [(makeWorldState i, makeWorldState j) | (i, j) <- intRel]
  in KrM worldStates newVal newRel

makeWorldState :: Integer -> WorldState
makeWorldState n =
  let uniqueAtom = PPred "WorldID" [StdName (show n)]
  in mkWorldState [(uniqueAtom, True)] []
```

To be able to print models, we define a `Show` instance for `KripkeModel a`:

```haskell
instance Show a => Show (KripkeModel a) where
   show (KrM uni val rel) = "KrM\n" ++ show uni ++ "\n" ++ show [(x, val x) | x <- uni ] ++
        "\n" ++ show rel
```

Later, we will want to compare models for equality; so we'll also define an ?Eq? instance. Comparison for equality will work, at least as long as models are finite. The way this comparison works is by checking that the valuations agree on all worlds in the model. By sorting before checking for equality, we ensure that the order in which worlds appear in the list of worlds representing the universe, the order in which true propositions at a world appear, and the order in which pairs appear in the relation doesn't affect the comparison.

```haskell
instance (Eq a, Ord a) => Eq (KripkeModel a) where
   (KrM u v r) == (KrM u' v' r') =
      (nub. sort) u == (nub. sort) u' && all (\w -> (nub. sort) (v w) ==  (nub. sort) (v' w
          )) u && (nub. sort) r == (nub. sort) r'
```

NB: the following is possible: Two models of type `KripkeModel WorldStates` are equal, we convert both to models of type `KripkeModel Integers` and the resulting models are not equal.

Why is this possible? Because when checking for equality between models of type `KripkeModel WorldStates` we ignore the order of worlds in the list that defines the universe; but for the conversion to `KripkeModel Integer`, the order matters!

3.2 Translation functions: KL to Kripke

In our implementation, to do justice to the the fact that translation functions can only sensibly be defined for *some* Kripke models, and *some* $\mathcal{KL}$ formulas, we use the `Maybe` monad provided by Haskell.
To do justice to the fact that evaluating in a $\mathcal{KL}$-model is more like evaluating a formula at a specific world in a Kripke model, than like evaluating a formula with respect to a whole Kripke model, we translate from pairs of Kripke models and worlds to $\mathcal{KL}$-models, rather than just from Kripke models to $\mathcal{KL}$-models.
Thus, these are the types of our translation functions:

1. `translateFormToKr :: Formula -> Maybe ModForm`
2. `translateFormToKL :: ModForm -> Formula`
3. `translateModToKr :: Model -> KripkeModel WorldState`
4. `kripkeToKL :: KripkeModel WorldState -> WorldState -> Maybe Model`

What constraints do we want our translation functions to satisfy? We propose that reasonable translation functions should at least satisfy these constraints: for any $\mathcal{KL}$ model `Model w e d`, any translatable $\mathcal{KL}$ formula `f`, any translatable Kripke model `KrM uni val rel`, and any modal formula `g`,

1. Translating formulas back and forth shouldn't change them:
   - `translateFormToKL (fromJust (translateFormToKr f)) = f`
   - `fromJust (translateFormToKr (translateFormToKL g)) = g`
2. Truth values should be preserved by the translations:
   - `Model w e d |= f` iff
     `(translateModToKr (Model w e d)) w |= fromJust (translateFormToKr f)`
   - `(KrM uni val rel) w |= g` iff
     `fromJust (kripkeToKL (KrM uni val rel) w) |= translateFormToKL g`

We check that our translation formulas do indeed satisfy these constraint in the test suite (in `TranslatorSpec.lhs`).

3.3 Translation Functions from $\mathcal{KL}$ to Kripke

Translation Functions for Formulas
As mentioned above, we translate from a fragment of the language of $\mathcal{KL}$ to the language of propositional modal logic. Specifically, only formulas whose atomic subformulas consist of the

predicate letter "P", followed by exactly one standard name, are translated; in this case the function `translateFormToKr` replaces all of the atomic subformulas by propositional variables.

```
translateFormToKr :: Formula -> Maybe ModForm
translateFormToKr (Atom (Pred "P" [StdNameTerm (StdName nx)])) = Just $ P (read (drop 1 nx)
    )
translateFormToKr (Not f)                           =  Neg <$> translateFormToKr f
translateFormToKr (Or f g)                          = fmap Dis (translateFormToKr f) <*>
    translateFormToKr g
translateFormToKr (K f)                             = Box <$> translateFormToKr f
translateFormToKr _                                 = Nothing
```

Translation Functions for Models

`translateModToKr` takes a $\mathcal{KL}$ model, and returns a Kripke model, where

- the worlds are all the world states in the epistemic state of the $\mathcal{KL}$ model, plus the actual world state;
- for each world, the propositional variables true at it are the translations of the atomic formulas consisting of "P" followed by a standard name that are true at the world state;
- the worlds from within the epistemic state all see each other, and themselves and the actual world sees all other worlds.

```
translateModToKr :: Model -> KripkeModel WorldState
translateModToKr (Model w e _) = KrM (nub (w:Set.toList e)) val (nub rel) where
   val = trueAtomicPropsAt
   rel = [(v, v') | v <- Set.toList e, v' <- Set.toList e] ++ [(w,v) | v <- Set.toList e]

--the next two are helper functions:
--identifies true atomic formulas at a world that consist of the predicate "P" followed by
    a standard name
trueAtomicPropsAt :: WorldState -> [Proposition]
trueAtomicPropsAt w =
   map actualAtomToProp trueActualAtoms where
      trueActualAtoms = filter isActuallyAtomic $ map fst (filter snd (Map.toList (
          atomValues w)))
      actualAtomToProp :: Atom -> Proposition
      actualAtomToProp (Pred "P" [StdNameTerm (StdName nx)]) = read (drop 1 nx)
      actualAtomToProp _ = error "actualAtomToProp should only be given atoms of the form '
          P(standardname)' as input"

--checks whether an atomic formula consists of the predicate "P" followed by a standard
    name
isActuallyAtomic :: Atom -> Bool
isActuallyAtomic (Pred "P" [StdNameTerm (StdName _)]) = True
isActuallyAtomic _ = False
```

3.4 Translating from Propositional Modal Logic to $\mathcal{KL}$

Translation Functions for Formulas

`translateFormToKL` takes a formula of propositional modal logic and computes the translated $\mathcal{KL}$ formula. Since PML is a propositional logic, we will immitate this in the language of $\mathcal{KL}$ by translating it to a unique corresponding atomic formula in $\mathcal{KL}$.

```
-- Translates an a formula of propositional modal logic to a KL formula (predicate logic
    with knowledge operator).
translateFormToKL :: ModForm -> Formula
translateFormToKL (P n) = Atom (Pred "P" [StdNameTerm (StdName ("n" ++ show n))])  -- Maps
    proposition P n to atom P(n), e.g., P 1 -> P(n1)
```

```
translateFormToKL (Neg form) = Not (translateFormToKL form)                    --
    Negation is preserved recursively
translateFormToKL (Dis form1 form2) = Or (translateFormToKL form1) (translateFormToKL form2
    )
translateFormToKL (Box form) = K (translateFormToKL form)                      -- Box
     becomes K, representing knowledge
```

Translation Functions for Models
`kripkeToKL` takes a Kripke model and a world in its universe and computes a corresponding $\mathcal{KL}$-model which is satisfiability equivalent with the given world in the given model.

$\mathcal{KL}$ models and Kripke Models can both be used to represent an agent's knowledge, but they do it in a very different way. A $\mathcal{KL}$ model (e, w) is an ordered pair of a *world state* w , representing what is true in the real world, and an *epistemic state* e, representing what the agent considers possible.
In contrast, a Kripke model $\mathcal{M} = (W, R, V)$ consists of a universe W, an accessibility relation $R \subseteq W \times W$, and a valuation function $V : Prop \to \mathcal{P}(W)$ that assigns each propositional letter the set of worlds in which it is true.
There are two key differences between $\mathcal{KL}$ models and Kripke Models. First, $\mathcal{KL}$ models have a fixed actual world and can only evaluate non-modal formulas at this particular world while Kripke Models can evaluate what is true at each of the worlds in their Universe. Second, the *world states* in the *epistemic state* of a $\mathcal{KL}$ model form an equivalence class in the sense that no matter how many nested *K-Operators* there are in a formula, each level is evaluated on the whole epistemic state. Among others, this implies that positive introspection ($\mathbf{K}\varphi \to \mathbf{KK}\varphi$) and negative introspection ($\neg\mathbf{K}\varphi \to \mathbf{K}\neg\mathbf{K}\varphi$) are valid in $\mathcal{KL}$. Informally, positive introspection says that if an agent knows φ, then they know that they know φ and negative introspection says that if an agent does not know φ, then they know that they do not know φ. In Kripke models, however, this is not the case and the worlds accessible from each world do not always form an equivalence class under the accessibility relation R.

We address the first difference by not translating the entire Kripke Model but by selecting an actual world in the Kripke model and then translating the submodel point generated at this world into a $\mathcal{KL}$ model. By design, the selected actual world is translated to the actual *world state* and the set of worlds accessible from the selected world is translated to the *epistemic state*. Further, we restrict the translation function to only translate the fragment of Kripke Models where the set of worlds accessible from each world in the universe form an equivalence class with respect to R.

This gives us a translation function `kripkeToKL` of type
`kripkeToKL :: KripkeModel WorldState -> WorldState -> Maybe Model`

Constraints on Translatable Kripke Models
To ensure that the set of worlds accessible from each world in the universe form an equivalence class with respect to R, we require the Kripke model to be transitive ($\forall u, v, w((Ruv \land Rvw) \to Ruw))$ and euclidean ($\forall u, v, w((Ruv \land Ruw) \to Rvw))$.
For this, we implemented the following two functions that check whether a Kripke model is transitive and euclidean, respectively:

```
-- Checks if a Kripke model is Euclidean
isEuclidean :: (Eq a, Ord a) => KripkeModel a -> Bool
```

```
isEuclidean = isValidKr (Dis (Box (Neg (P 1))) (Box (dia (P 1))))  -- \Box \neg P1 \lor \
    Box \Diamond P1 holds for Euclidean relations

-- Checks if a Kripke model is transitive
isTransitive :: (Eq a, Ord a) => KripkeModel a -> Bool
isTransitive = isValidKr (Dis (Neg (Box (P 1))) (Box (Box (P 1))))  -- \neg \Box P1 \lor \
    Box \Box P1 holds for transitive relations
```

We further need the constraint that the world selected to be the actual world in the Kripke Model is in the universe of the given Kripke Model. This is ensured by the `isInUniv` function.

```
-- Checks if a world is in the Kripke models universe
isInUniv :: WorldState -> [WorldState] -> Bool
isInUniv = elem  -- Simple membership test
```

Main Function to Translate Kripke Models

With this, we can now define the `kripkeToKL` function that maps a Kripke Model of type `KripkeModel WorldState` and a `WorldState` to a `Just` $\mathcal{KL}$? `model` if the Kripke Model is transitive and euclidean and the selected world state is in the universe of the Kripke Model and to `Nothing` otherwise.

```
-- Main function: Convert Kripke model to KL model
kripkeToKL :: KripkeModel WorldState -> WorldState -> Maybe Model
kripkeToKL kr@(KrM univ val rel) w
  | not (isEuclidean kr && isTransitive kr) || not (isInUniv w univ) = Nothing
  | otherwise = Just (Model newWorldState newEpistemicState newDomain)
  where
    -- New actual world based on valuation of w
    newWorldState = createWorldState (val w)
    -- Accessible worlds from w
    accessibleWorlds = [v | (u, v) <- rel, u == w]
    -- New epistemic state: one WorldState per accessible world
    newEpistemicState = Set.fromList [createWorldState (val v) | v <- accessibleWorlds]
    -- Domain (empty for simplicity)
    newDomain = Set.empty
```

4 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
```

```haskell
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: A `Node` pairs formulas with world identifiers, and a `Branch` tracks nodes and used parameters.

```haskell
-- 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).

```haskell
-- 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

The function `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 a list of nodes, `nodes :: [Node]` (each `Node f w` is a formula `f` in world `w`), and a list of used parameters, `params :: Set StdName`. 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,` 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`. It 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 functions we use `isSatisfiable` and `isValid`. The function `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 functions `isSatisfiable`, `expandTableau`, and `isClosed` interact as follows: `isSatisfiable` starts the process with a single branch containing the formula. Then, `expandTableau` recursively applies `applyRule` to decompose formulas, creating new branches as needed (e.g., for $\lor$, $\exists$). In a next step, `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))
```

4.1 Tests

You can run all the tests and examine the current code coverage run `stack clean && stack test --covera`

```
{-# LANGUAGE InstanceSigs #-}

module Generators where

import SyntaxKL (Term(..), Atom(..), Formula(..), StdName)
import SemanticsKL
import Translator

import Data.Set (Set)
import qualified Data.Set as Set
import Data.List
import qualified Data.Map as Map
import Test.QuickCheck
```

This file contains helper generators, used only in testing. s

```
-- Generator for arbitrary upper case letter
genUpper :: Gen String
```

```
genUpper = (:[]) <$> elements ['A'..'Z']

-- Generator for arbitrary lower case letter
genLower :: Gen String
genLower = (:[]) <$> elements ['a'..'z']

-- Generator for ground terms
genGroundTerm :: Gen Term
genGroundTerm = sized genTerm
    where
        genTerm 0 = StdNameTerm <$> arbitrary
        genTerm n = oneof [StdNameTerm <$> arbitrary,
                           FuncAppTerm <$> genLower <*> resize (n `div` 2) (listOf
                               genGroundTerm)]

-- Generator for ground Atoms
genGroundAtom :: Gen Atom
genGroundAtom = Pred <$> genUpper<*> listOf genGroundTerm

-- Generator for ground formulas
genGroundFormula :: Gen Formula
genGroundFormula = sized genFormula
    where
        genFormula 0 = Atom <$> genGroundAtom
        genFormula n = oneof [ Atom <$> genGroundAtom
                             , Equal <$> genGroundTerm <*> genGroundTerm
                             , Not <$> genFormula (n `div` 2)
                             , Or <$> genFormula (n `div` 2) <*> genFormula (n `div` 2)
                             , K <$> genFormula (n `div` 2)
                             ]

-- Generator for a set of StdName values
genStdNameSet :: Gen (Set StdName)
genStdNameSet = sized $ \n -> do
  let m = min n 5
  size <- choose (0, m)
  Set.fromList <$> vectorOf size arbitrary

-- Helper Generator for atoms of the form 'P(standardname)'
genPAtom :: Gen Atom
genPAtom =  do
  n <- arbitrary
  return (Pred "P" [StdNameTerm n])

-- Generator for KL-formulas that can be translated
genTransFormula :: Gen Formula
genTransFormula = sized genTrForm
    where
        genTrForm 0 = Atom <$> genPAtom
        genTrForm n = oneof [ Atom <$> genPAtom
                            , Not <$> genTrForm (n `div` 2)
                            , Or <$> genTrForm (n `div` 2) <*> genTrForm (n `div` 2)
                            , K <$> genTrForm (n `div` 2)
                            ]


-- Generator for SEL-Formulae
genModForm :: Gen ModForm
genModForm = sized genFormula
  where
    genFormula :: Int -> Gen ModForm
    genFormula 0 = P <$> choose (1, 5)  -- Base case: atomic proposition
    genFormula n = frequency
      [ (2, P <$> choose (1, 5))                 -- Atomic proposition
      , (1, Neg <$> genFormula (n `div` 2))      -- Negation
      , (1, Dis <$> genFormula (n `div` 2)       -- Disjunction
                <*> genFormula (n `div` 2))
      , (1, Box <$> genFormula (n `div` 2))      -- Box operator
      ]

-- Generator for WorldStates at which only propositions of the form 'P(standardname)' are
    true
genTransWorldState ::Gen WorldState
```

```haskell
genTransWorldState = do
    let m = 5
    atoms <- vectorOf m genPAtom `suchThat` (\xs -> nub xs == xs)
    tvs <- vectorOf m arbitrary
    let atValues' = zip atoms tvs
    let atValues = Map.fromList $ checkDups atValues'
    WorldState atValues <$> arbitrary

-- Generator for smaller transitive and Euclidean Kripke models
genSmallTransEucKripke :: Gen (KripkeModel WorldState)
genSmallTransEucKripke = resize 6 genTransEucKripke

-- Generator for transitive and Euclidean Kripke models
genTransEucKripke :: Gen (KripkeModel WorldState)
genTransEucKripke = sized randomModel where
    randomModel :: Int -> Gen (KripkeModel WorldState)
    randomModel n = do
      msize <- choose (1, 1+n)
      u <- nub . sort <$> vectorOf msize genTransWorldState
      let v = trueAtomicPropsAt
      r' <- if null u
        then return []
        else listOf $ do
          x <- elements u
          y <- elements u
          return (x,y)
      let r = transEucClosure r'
      return (KrM u v r)

-- Generator for arbitrary world state Kripke models
genRandomKripkeModel :: Gen (KripkeModel WorldState)
genRandomKripkeModel = sized randomModel where
    randomModel :: Int -> Gen (KripkeModel WorldState)
    randomModel n = do
      msize <- choose (1, 1+n)
      u <- nub . sort <$> vectorOf msize genTransWorldState
      let v = trueAtomicPropsAt
      r' <- if null u
        then return []
        else listOf $ do
          x <- elements u
          y <- elements u
          return (x,y)
      return (KrM u v r')

genSmallRandomucKripke :: Gen (KripkeModel WorldState)
genSmallRandomucKripke = resize 6 genRandomKripkeModel


genIntKripkeModel :: Gen (KripkeModel Integer)
genIntKripkeModel = do
  n <- choose (1, 6::Integer)  -- Small universe for simplicity
  let univ = [0 .. n-1]
  val <- vectorOf (fromInteger n) (sublistOf [1..4]) >>= \props -> return $ \w -> props !!
      fromInteger w
  rel <- sublistOf [(w, w') | w <- univ, w' <- univ]
  return $ KrM univ val rel

transEucClosure :: Eq a => [(a,a)] -> [(a,a)]
transEucClosure rela
    | rela == closure   = rela
    | otherwise              = transEucClosure closure where
    closure = nub $ rela ++ [(a,c) | (a,b) <- rela, (b',c) <- rela, b == b'] ++ [(b,c) | (a
        ,b) <- rela, (a',c) <- rela, a == a']

-- Generator, which, given a KripkeModel, picks a world
genWorldFrom :: (KripkeModel a) -> Gen (World a)
genWorldFrom m = elements (universe m)

-- Generator for a pair consisting of a transitive, Euclidean model, and a world in that
   model
genTransEucKripkeWithWorld :: Gen (KripkeModel WorldState, World WorldState)
genTransEucKripkeWithWorld = do
```

```
  m <- genSmallTransEucKripke
  w <- genWorldFrom m
  return (m, w)

genNonEmptyEpistemicState :: Gen (Set WorldState)
genNonEmptyEpistemicState = do
    ws <- listOf1 arbitrary  -- `listOf1` ensures at least one WorldState
    return (Set.fromList ws)
```

```
module AskTellSpec where

import Test.Hspec


import AskTell
import SyntaxKL
import SemanticsKL
import Generators

import Test.QuickCheck
import qualified Data.Map as Map
import Data.Set (Set)
import qualified Data.Set as Set


spec :: Spec
spec = describe "ask - Example Tests" $ do
        let atom = Pred "P" []
            f = Atom atom
            ask_ws1 = WorldState (Map.fromList [(atom, True)]) Map.empty
            ask_ws2 =  WorldState (Map.fromList [(atom, False)]) Map.empty
            d = Set.empty
        it "ask returns False when the epistemic state is empty" $ do
                let e = Set.empty
                ask d e f `shouldBe` False
        it "ask returns False when formula is not True in all world states." $ do
                let e = Set.fromList [ask_ws1,ask_ws2]
                ask d e f `shouldBe` False
        it "ask returns True when formula is True in all world states." $ do
                let e = Set.fromList [ask_ws1]
                ask d e f `shouldBe` True
        -- ask vs askModel
        it "askModel = ask = False when the epistemic state is empty" $ do
                let e = Set.empty
                    m = Model {actualWorld = (Set.findMin e), epistemicState = e, domain =d
                        }
                ask d e f `shouldBe` askModel m f
        it "askModel = ask = False when formula is not True in all world states." $ do
                let e = Set.fromList [ask_ws1,ask_ws2]
                    m = Model {actualWorld = (Set.findMin e), epistemicState = e, domain =d
                        }
                ask d e f `shouldBe` askModel m f
        it "askModel = ask = True when formula is True in all world states." $ do
                let e = Set.fromList [ask_ws1]
                    m = Model {actualWorld = (Set.findMin e), epistemicState = e, domain =d
                        }
                ask d e f `shouldBe` askModel m f

        describe "ask - Property Tests" $ do
                it "ask is true for the tautology P(x) -> ~~ P(x)" $ do
                        let taut1 = (Or (Not f) (Not (Not f)))
                        property $
                                forAll genStdNameSet $ \d' ->  -- d' can be any set
                                forAll genNonEmptyEpistemicState $ \e ->  -- Ensure e is
                                    non-empty
                                        ask (d' :: Set StdName) (e :: Set WorldState) taut1
                                             `shouldBe` True
                it "ask is true for the tautology P or ~P" $ do
                        let taut2 = Or (f) (Not f)
                        property $
                                forAll genStdNameSet $ \d' ->  -- d' can be any set
                                forAll genNonEmptyEpistemicState $ \e ->  -- Ensure e is
```

```
                                          non-empty
                                              ask (d' :: Set StdName) (e :: Set WorldState) taut2
                                                   `shouldBe` True
        -- TELL
        describe "tell - Example Tests" $ do
                let atom1 = Pred "P1" []
                    atom2 = Pred "P2" []
                    tell_ws1 = WorldState (Map.fromList [(atom1, True), (atom2, True)]) Map
                        .empty
                    tell_ws2 =  WorldState (Map.fromList [(atom1, True), (atom2, False)])
                        Map.empty
                    e' = Set.fromList [tell_ws1, tell_ws2]
                    d' = Set.empty
                    f' = Atom atom1
                    g = Atom atom2
                    m = Model {actualWorld = (Set.findMin e'), epistemicState = e', domain
                        =d'}
                it "tell returns the same epistemic state when formula is known" $ do
                        tell d' e' f' `shouldBe` e'
                it "tell returns different epistemic state when formula is not known" $ do
                        (tell d e' g /= e') `shouldBe` True
               -- tell vs tellModel
                it "tell == tellModel  when formula is known" $ do
                        (tell d' e' f' == epistemicState (tellModel m f') ) `shouldBe` True
                it "tell == tellModel  when formula is not known" $ do
                        (tell d' e' g == epistemicState (tellModel m g)) `shouldBe` True
        describe "tell - Property Tests" $ do
                let f' = Atom (Pred "P1" [])
                it "tell shouldnt restrict the epistemic state for the tautology P(x) -> ~~
                    P(x)" $ do
                        let taut1 = (Or (Not f') (Not (Not f')))
                        property $ \e ->
                            forAll genStdNameSet $ \d' ->
                                tell  (d' :: Set StdName) (e :: Set WorldState) taut1 `
                                    shouldBe` e

        -- Initial
        describe "initial - Example Tests" $ do
                let patoms_empt = []
                    pterms_empt = []
                    snames_empt = []
                    patoms = [PPred "P1" [StdName "S1", StdName "S2"],PPred "P2" [StdName "
                        S2", StdName "S3"]]
                    pterms = [PStdNameTerm (StdName "S1"),PStdNameTerm (StdName "S2"),
                        PStdNameTerm (StdName "S3")]
                    snames = [StdName "S1", StdName "S2", StdName "S3"]
                it "initial with empty inputs produces empty epistemic state" $ do
                        initial patoms_empt pterms_empt snames_empt == Set.empty `shouldBe`
                             True
                it "initial with nonempty inputs should be nonempty" $ do
                        initial patoms pterms snames /= Set.empty `shouldBe` True

        -- Poteential tests to be added:
        -- initial - tell (contradiction) - is empty
        -- ask (tell e f) f - always returns true
```

```
module SemanticsKLSpec where

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

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

import SemanticsKL -- tested module
import SyntaxKL    -- types used in tests
import Generators  -- helper functions for testing
```

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.

```
spec :: Spec
spec =  describe "evalTerm - Unit Tests" $ do
        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 "evalTerm - Property Tests" $ do
            it "evalTerm errors for all variables passed" $ do
                property $ \ w x -> evaluate (evalTerm w (VarTerm x)) `shouldThrow`
                    anyException
            it "evalTerm returns the StdName for StdNameTerm" $ do
                property $ \w n -> evalTerm w (StdNameTerm n) == n


        describe "isGround - Unit Tests" $ do
            it "isGround returns True for StdNameTerm" $ do
                isGround (StdNameTerm $ StdName "n1") `shouldBe` True
            it "isGround returns False for VarTerm" $ do
                isGround (VarTerm $ Var "x") `shouldBe` False
            it "isGround returns True for FuncAppTerm with all ground arguments" $ do
                isGround (FuncAppTerm "f" [StdNameTerm $ StdName "n1"]) `shouldBe` True
            it "isGround returns False for complex FuncAppTerm with at least one non-ground
             argument" $ do
                let term = FuncAppTerm "f" [FuncAppTerm "g" [VarTerm $ Var "x", StdNameTerm
                     $ StdName "n1"]]
                isGround term `shouldBe` False


        describe "isGroundFormula - Unit Tests" $ do
            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 "isGroundFormula - Property Tests" $ do
            it "isGroundFormula returns True for groundFormula" $ do
                property $ forAll genGroundFormula $ \f -> isGroundFormula (f :: Formula)
            --todo fix this test
            it "isGroundFormula returns False for Exists" $ do
                property $ \n f -> not $ isGroundFormula (Exists (Var n) (f :: Formula))


        describe "substTerm - Unit Tests" $ 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 - Unit Tests" $ 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 - Property Tests" $ do
    -- test fixtures
    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 - Unit Tests" $ do
        -- test fixtures
        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 - Property Tests" $ do
        -- This is an expensive test, so we limit the size of the formula
        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 - Property Tests" $ do
        -- test fixtures
        let x = Var "x"
            px = Atom (Pred "P" [VarTerm x])
        describe "checkModel satisfies validities when atoms are unground" $ do
            it "checkModel arbitrary model satisfies P(x) -> ~~ P(x)" $ do
                property $ \m -> checkModel m (Or (Not px) (Not (Not px))) `shouldBe`
                    True
            it "checkModel  arbitrary model satisfies for x=x" $ do
                property $ \m -> checkModel m (Equal (VarTerm x) (VarTerm x)) `shouldBe
                    ` True
```

```
module SyntaxKLSpec where

import Test.Hspec
import SyntaxKL

spec :: Spec
spec = describe "func_name1 - Specify collection of Unit/Property/Integration Tests" $ do
            it "test description" $
                True `shouldBe` True
```

```
module TableauSpec where

import Test.Hspec
import Tableau
```

```
spec :: Spec
spec = describe "func_name1 - Specify collection of Unit/Property/Integration Tests" $ do
            it "test description" $
                True `shouldBe` True
```

- LANGUAGE NondecreasingIndentation -

```
module TranslatorSpec where

import Test.Hspec
import Test.QuickCheck
import Data.Maybe
import Data.Set (Set)
import qualified Data.Set as Set


import Translator hiding (dia)
import SyntaxKL
import SemanticsKL

import Generators

-- We will use the following repeatedly, and therefore define them globally.
n1, n2, n3, n4 :: StdName
n1 = StdName "n1"
n2 = StdName "n2"
n3 = StdName "n3"
n4 = StdName "n4"

spec :: Spec
spec =  describe "translateFormToKr" $ do
        let formula1 = Atom (Pred "P" [StdNameTerm n1])
            formula2 = K (Atom (Pred "P" [StdNameTerm n1]))
            formula3 = Not (K (Atom (Pred "P" [StdNameTerm n1])))
            trFormula1 = P 1
            trFormula2 = Box (P 1)
            trFormula3 = Neg (Box (P 1))
            formula4 = Atom (Pred "Teach" [StdNameTerm n1, StdNameTerm n2])
            formula5 = Not (K (Atom (Pred "Q" [StdNameTerm n1])))

        it "'P(n1)' correctly translated" $ do
            fromJust (translateFormToKr formula1) `shouldBe` trFormula1
        it "'K(P(n1))' correctly translated" $ do
            fromJust (translateFormToKr formula2) `shouldBe` trFormula2
        it "'~K(P(n1))' correctly translated" $ do
            fromJust (translateFormToKr formula3) `shouldBe` trFormula3
        it "'Teach(n1, n2)' shouldn't be translated (the translation function should return
             Nothing)" $ do
            isNothing (translateFormToKr formula4) `shouldBe` True
        it "'~K(Q(n1))' shouldn't be translated (the translation function should return
            Nothing)" $ do
            isNothing (translateFormToKr formula5) `shouldBe` True

        describe "translateModToKr" $ do
            let -- a model where the actual world is part of the epistemic state
                w1, w2, w3, w4 :: WorldState
                w1 = mkWorldState [ (PPred "P" [n1], True) ] []
                w2 = mkWorldState [ (PPred "P" [n2], True)
                     , (PPred "P" [n3], True) ] []
                w3 = mkWorldState [ (PPred "P" [n4], True) ] []
                w4 = mkWorldState [] []

                e1 :: EpistemicState
                e1 = Set.fromList [w1, w2, w3, w4]

                domain1 :: Set StdName
                domain1 = Set.fromList [n1, n2, n3, n4]

                model1 :: Model
                model1 = Model w1 e1 domain1

                -- if all goes well, this should be converted to the following KripkeModel
```

```
            kripkeM1 :: KripkeModel WorldState
            kripkeM1 = KrM uni val rel where
                uni = [w1, w2, w3, w4]
                val world    | world == w1 = [1]
                             | world == w2 = [2, 3]
                             | world == w3 = [4]
                             | otherwise   = []
                rel = [(v, v') | v <- uni, v' <- uni]

            -- a model that contains non-atomic formulas
            w2' :: WorldState
            w2' = mkWorldState [ (PPred "P" [n2], True)
                               , (PPred "P" [n3], True)
                               , (PPred "R" [n4, n1], True)] []

            e1' :: EpistemicState
            e1' = Set.fromList [w1, w2', w3, w4]

            -- model1' is like model1, except for extra formulas true in w2' that aren'
                t
            -- true in w2
            model1' :: Model
            model1' = Model w1 e1' domain1

        it "correct conversion of model where actual world is in epistemic state" $ do
            translateModToKr model1 == kripkeM1 `shouldBe` True
        it "correct conversion of model where actual world is not in epistemic state" $
             do
            translateModToKr model1 == kripkeM1 `shouldBe` True
        it "if the only difference between two KL models is in one of them, additional
            formulas not of the form 'P(standardname)' are true, they should be
            converted to Kripke models that 'look the same'" $ do
            translateKrToKrInt (translateModToKr model1) == translateKrToKrInt (
                translateModToKr model1') `shouldBe` True

    describe "integration tests language translation" $ do
        it "translating KL to SEL formula, and then back, shouldn't change anything" $
            do
            property $ forAll genTransFormula $ \f -> f == translateFormToKL (fromJust
                (translateFormToKr f))
        it "translating SEL to KL, and then back, shouldn't change anything" $ do
            property $ \g -> g == fromJust (translateFormToKr (translateFormToKL g))

    describe "combined" $ do
        it "if a translatable formula is true in a KL model, its translation should be
            true at the corresponding Kripke model and world, and vice versa" $ do
            property $ forAll (do
                m <- arbitrary
                f <- genTransFormula
                return (m, f)
                ) $ \(Model w e d, f) -> (Model w e d `satisfiesModel` f) <==> ((
                    translateModToKr (Model w e d), w) `makesTrue` fromJust (
                    translateFormToKr f))

        it "if a formula is true at a world and Kripke model, then it should be true at
             the corresponding KL model, and vice versa" $ do
            property $ forAll (do
                (m, w) <- genTransEucKripkeWithWorld `suchThat` (\(m, w) -> w `elem`
                    universe m)
                g <- arbitrary
                return (m, w, g)
                ) $ \(m, w, g) ->  (m, w) `makesTrue` g <==> (fromJust (kripkeToKL m w)
                     `satisfiesModel` translateFormToKL g)

    describe "trueEverywhere" $ do
        let ww1 = mkWorldState [(PPred "P" [StdName "n1"], True)] []
            ww2 = mkWorldState [(PPred "P" [StdName "n1"], True), (PPred "P" [StdName "
                n2"], True)] []
            modelAllP1 = KrM [ww1, ww2] trueAtomicPropsAt [(ww1, ww1), (ww1, ww2), (ww2
                , ww1), (ww2, ww2)]
            modelSomeP1 = KrM [ww1, mkWorldState [] []] trueAtomicPropsAt [(ww1, ww1)]

            w1, w2 :: WorldState
```

```
        w1 = mkWorldState [ (PPred "P" [n1], True) ] []
        w2 = mkWorldState [ (PPred "P" [n2], True)
            , (PPred "P" [n3], True) ] []

    it "returns true when P 1 is true in every world" $ do
        trueEverywhere modelAllP1 (P 1) `shouldBe` True

    it "returns false when P 1 is not true in every world" $ do
        trueEverywhere modelSomeP1 (P 1) `shouldBe` False

    it "returns true for Box (P 1) when P 1 is true in all accessible worlds from
        every world" $ do
        trueEverywhere modelAllP1 (Box (P 1)) `shouldBe` True

    it "returns false for Box (P 1) when P 1 is not true in all accessible worlds"
        $ do
        let modelPartial = KrM [w1, w2] trueAtomicPropsAt [(w1, w2)]
        trueEverywhere modelPartial (Box (P 1)) `shouldBe` False

    it "holds everywhere if and only if makesTrue holds for all worlds (property)"
        $
        property $ do
            model <- genRandomKripkeModel
            formula <- genModForm
            let holdsEverywhere = trueEverywhere model formula
                univ = universe model
            return $ holdsEverywhere <==> all (\w -> makesTrue (model, w) formula)
                univ

    it "Box f holds everywhere iff f holds in all accessible worlds from every
        world (property)" $
        property $ do
            model <- genRandomKripkeModel
            f <- genModForm
            let univ = universe model
                rel = relation model
                holdsEverywhere = trueEverywhere model (Box f)
                allAccessibleSatisfy = all (\w -> all (\w' -> makesTrue (model, w')
                     f) (rel ! w)) univ
            return $ holdsEverywhere <==> allAccessibleSatisfy



describe "kripkeToKL" $ do
-- Specific test cases
    let ww1 = mkWorldState [(PPred "P" [StdName "n1"], True)] []
        ww2 = mkWorldState [(PPred "P" [StdName "n2"], True)] []
        ww3 = mkWorldState [(PPred "P" [StdName "n3"], True)] []
        -- Non-Euclidean model: ww1 -> ww2, ww1 -> ww3, but ww2 -/-> ww3
        nonEuclideanKr = KrM [ww1, ww2, ww3] trueAtomicPropsAt [(ww1, ww2), (ww1,
            ww3)]
        -- Non-transitive model: ww1 -> ww2 -> ww3, but ww1 -/-> ww3
        nonTransitiveKr = KrM [ww1, ww2, ww3] trueAtomicPropsAt [(ww1, ww2), (ww2,
            ww3)]
        -- Valid model but w not in universe
        validKr = KrM [ww1, ww2] trueAtomicPropsAt [(ww1, ww1), (ww1, ww2), (ww2,
            ww1), (ww2, ww2)]
        wOut = mkWorldState [(PPred "P" [StdName "n4"], True)] []

    it "returns Nothing for a non-Euclidean model" $ do
        isNothing (kripkeToKL nonEuclideanKr ww1) `shouldBe` True

    it "returns Nothing for a non-transitive model" $ do
        isNothing (kripkeToKL nonTransitiveKr ww1) `shouldBe` True

    it "returns Nothing when the world is not in the universe" $ do
        isNothing (kripkeToKL validKr wOut) `shouldBe` True

    it "returns Nothing iff not (isEuclidean && isTransitive) || not (isInUniv w
        univ) (property)" $
        property $ do
            kr@(KrM univ _ _) <- genSmallRandomucKripke
            w <- oneof ([elements univ | not (null univ)] ++ [genTransWorldState])
```

```
            let result = kripkeToKL kr w
                condition = not (isEuclidean kr && isTransitive kr) || not (
                    isInUniv w univ)
            return $ isNothing result <==> condition

describe "convertToWorldStateModel" $ do
    let intModel = KrM [0, 1] (\w -> if w == 0 then [1] else [2]) [(0, 1)]
        expectedKr = KrM [makeWorldState 0, makeWorldState 1]
            (\w -> if w == makeWorldState 0 then [1] else [2])
            [(makeWorldState 0, makeWorldState 1)]

    it "correctly converts an IntKripkeModel to a KripkeModel" $ do
        let converted = convertToWorldStateModel intModel
        universe converted `shouldBe` universe expectedKr
        all (\w -> valuation converted w == valuation expectedKr w) (universe
            converted) `shouldBe` True
        relation converted `shouldBe` relation expectedKr

    it "preserves structure in conversion (property)" $
        property $ do
        intKr <- genIntKripkeModel
        let kr = convertToWorldStateModel intKr
            KrM intUniv intVal intRel = intKr
            worldStates = map makeWorldState intUniv
        return $ length (universe kr) == length intUniv &&
            relation kr == [(makeWorldState i, makeWorldState j) | (i, j) <- intRel
                ] &&
            all (\(i, w) -> valuation kr w == intVal i) (zip intUniv worldStates)

    it "produces distinct WorldStates for distinct integers (property)" $
        property $ do
            n <- choose (0, 100)
            m <- choose (0, 100) `suchThat` (/= n)
            let ws1 = makeWorldState n
                ws2 = makeWorldState m
            return $ ws1 /= ws2

describe "Show KripkeModel Integer" $ do
    let intModel = KrM [0::Integer, 1] (\w -> if w == 0 then [1] else [2]) [(0, 1)]
    it "shows KripkeModel Integer in expected format" $ do
        show intModel `shouldBe` "KrM\n[0,1]\n[(0,[1]),(1,[2])]\n[(0,1)]"

    it "consistently formats universe, valuation, and relation (property)" $
        property $ do
            intKr@(KrM univ val rel) <- genIntKripkeModel
            let expected = "KrM\n" ++ show univ ++ "\n" ++ show [(x, val x) | x <-
                univ] ++ "\n" ++ show rel
            return $ show intKr == expected

describe "Show KripkeModel WorldState" $ do
    let krModel = KrM [makeWorldState 0, makeWorldState 1]
                    (\w -> if w == makeWorldState 0 then [1] else [2])
                    [(makeWorldState 0, makeWorldState 1)]
    it "shows KripkeModel WorldState via KripkeModel Integer conversion" $ do
        show (translateKrToKrInt krModel) `shouldBe` "KrM\n[0,1]\n[(0,[1]),(1,[2])
            ]\n[(0,1)]"

    it "translating KripkeModel Integer to KripkeModel WorldState and back show the
        same model" $
        property $ do
        show (convertToWorldStateModel (translateKrToKrInt krModel)) `shouldBe` "
            KrM\n[WorldState {atomValues = fromList [(Pred \"WorldID\" [StdNameTerm
             (StdName \"0\")],True)], termValues = fromList []},WorldState {
            atomValues = fromList [(Pred \"WorldID\" [StdNameTerm (StdName \"1\")],
            True)], termValues = fromList []}]\n[(WorldState {atomValues = fromList
             [(Pred \"WorldID\" [StdNameTerm (StdName \"0\")],True)], termValues =
            fromList []},[1]),(WorldState {atomValues = fromList [(Pred \"WorldID\"
             [StdNameTerm (StdName \"1\")],True)], termValues = fromList []},[2])]\
            n[(WorldState {atomValues = fromList [(Pred \"WorldID\" [StdNameTerm (
            StdName \"0\")],True)], termValues = fromList []},WorldState {
            atomValues = fromList [(Pred \"WorldID\" [StdNameTerm (StdName \"1\")],
            True)], termValues = fromList []})]"
```

```haskell
    describe "con" $ do
        it "computes conjunction as Neg (Dis (Neg f) (Neg g))" $ do
            con (P 1) (P 2) `shouldBe` Neg (Dis (Neg (P 1)) (Neg (P 2)))

        it "behaves like logical AND (property)" $
            property $ do
            kr <- genRandomKripkeModel
            f <- genModForm
            g <- genModForm
            let univ = universe kr
            return $ all (\w -> makesTrue (kr, w) (con f g) == (makesTrue (kr, w) f &&
                makesTrue (kr, w) g)) univ

    describe "impl" $ do
        it "computes implication as Dis (Neg f) g" $ do
            impl (P 1) (P 2) `shouldBe` Dis (Neg (P 1)) (P 2)

        it "behaves like logical implication (property)" $
            property $ do
            kr <- genRandomKripkeModel
            f <- genModForm
            g <- genModForm
            let univ = universe kr
            return $ all (\w -> makesTrue (kr, w) (impl f g) == (not (makesTrue (kr, w)
                 f) || makesTrue (kr, w) g)) univ


    describe "Translation (Propositional Modal Logic to KL)" $ do
        -- Helper functions
        let
            -- Example models (completed)
            intW0, intW1, intW2 :: World Integer
            intW0 = 0; intW1 = 1; intW2 = 2
            intUniverse1 = [intW0, intW1, intW2]
            intValuation1 w | w == 0 || w == 1 = [1] | w == 2 = [] | otherwise = []
            intRelation1 = [(0, 0), (0, 1), (1, 0), (1, 1), (2, 2)]
            intModel1 = KrM intUniverse1 intValuation1 intRelation1
            exampleModel1 = convertToWorldStateModel intModel1

            intW20, intW21, intW22 :: World Integer
            intW20 = 20; intW21 = 21; intW22 = 22
            intUniverse2 = [intW20, intW21, intW22]
            intValuation2 w | w == 20 = [1] | w == 21 = [] | w == 22 = [1] | otherwise
                = []
            intRelation2 = [(20, 21), (21, 22)]
            intModel2 = KrM intUniverse2 intValuation2 intRelation2
            exampleModel2 = convertToWorldStateModel intModel2

            intW30, intW31, intW32 :: World Integer
            intW30 = 30; intW31 = 31; intW32 = 32
            intUniverse3 = [intW30, intW31, intW32]
            intValuation3 w | w == 30 = [1] | otherwise = []
            intRelation3 = [(w1, w2) | w1 <- intUniverse3, w2 <- intUniverse3]
            intModel3 = KrM intUniverse3 intValuation3 intRelation3
            exampleModel3 = convertToWorldStateModel intModel3

            intW40, intW41, intW42 :: World Integer
            intW40 = 40; intW41 = 41; intW42 = 42
            intUniverse4 = [intW40, intW41, intW42]
            intValuation4 w | w == 40 = [1] | w == 41 = [] | w == 42 = [2] | otherwise
                = []
            intRelation4 = []
            intModel4 = KrM intUniverse4 intValuation4 intRelation4
            exampleModel4 = convertToWorldStateModel intModel4

            intW50, intW51, intW52 :: World Integer
            intW50 = 50; intW51 = 51; intW52 = 52
            intUniverse5 = [intW50, intW51, intW52]
            intValuation5 w | w == 50 = [1, 2] | w == 51 = [1] | w == 52 = [] |
                otherwise = []
            intRelation5 = [(50, 51), (51, 52), (52, 50)]
            intModel5 = KrM intUniverse5 intValuation5 intRelation5
            exampleModel5 = convertToWorldStateModel intModel5
```

```
    intW60, intW61, intW62, intW63, intW64 :: World Integer
    intW60 = 60; intW61 = 61; intW62 = 62; intW63 = 63; intW64 = 64
    intUniverse6 = [intW60, intW61, intW62, intW63, intW64]
    intValuation6 w | w == 60 = [1] | w == 61 = [1, 2] | w == 62 = [] | w == 63
         = [2] | w == 64 = [1, 2] | otherwise = []
    intRelation6 = let cluster1 = [60, 61, 62]; cluster2 = [63, 64]
                in [(w1, w2) | w1 <- cluster1, w2 <- cluster1] ++ [(w1, w2) |
                    w1 <- cluster2, w2 <- cluster2]
    intModel6 = KrM intUniverse6 intValuation6 intRelation6
    exampleModel6 = convertToWorldStateModel intModel6

    intW70, intW71, intW72, intW73, intW74, intW75 :: World Integer
    intW70 = 70; intW71 = 71; intW72 = 72; intW73 = 73; intW74 = 74; intW75 =
        75
    intUniverse7 = [intW70, intW71, intW72, intW73, intW74, intW75]
    intValuation7 w | w == 70 = [1] | w == 71 = [1, 2] | w == 72 = [2] | w ==
        73 = [] | w == 74 = [1] | w == 75 = [1, 2] | otherwise = []
    intRelation7 = let cluster = [71, 72, 73, 74, 75]
                in [(70, w) | w <- cluster] ++ [(w1, w2) | w1 <- cluster, w2 <-
                     cluster]
    intModel7 = KrM intUniverse7 intValuation7 intRelation7
    exampleModel7 = convertToWorldStateModel intModel7

    modelKL7 = fromJust (kripkeToKL exampleModel7 (makeWorldState 70))
    modelKL7b = fromJust (kripkeToKL exampleModel7 (makeWorldState 71))
    ref = translateFormToKL (impl (Box (P 99)) (P 99))

    smallModels :: [KripkeModel WorldState]
    smallModels = [exampleModel1, exampleModel2, exampleModel3, exampleModel4,
        exampleModel5, exampleModel6, exampleModel7]


it "translates atomic proposition P 1 correctly" $ do
    translateFormToKL (P 1) `shouldBe` Atom (Pred "P" [StdNameTerm (StdName "n1
        ")])

it "translates negation Neg (P 1) correctly" $ do
    translateFormToKL (Neg (P 1)) `shouldBe` Not (Atom (Pred "P" [StdNameTerm (
        StdName "n1")]))

it "translates conjunction con (P 1) (P 2) correctly" $ do
    translateFormToKL (con (P 1) (P 2)) `shouldBe`
        Not (Or (Not (Atom (Pred "P" [StdNameTerm (StdName "n1")])))
                (Not (Atom (Pred "P" [StdNameTerm (StdName "n2")]))))

it "translates box operator Box (P 1) correctly" $ do
    translateFormToKL (Box (P 1)) `shouldBe` K (Atom (Pred "P" [StdNameTerm (
        StdName "n1")]))

it "translates diamond operator dia (P 1) correctly" $ do
    translateFormToKL (dia (P 1)) `shouldBe`
        Not (K (Not (Atom (Pred "P" [StdNameTerm (StdName "n1")]))))

it "ensures invertibility of simple formula Box (P 1) using equivalence" $ do
    let g = Box (P 1)
        klForm = translateFormToKL g
        selForm = fromJust (translateFormToKr klForm)
    areEquivalent smallModels g selForm `shouldBe` True

it "ensures invertibility of complex formula Box (con (P 1) (Neg (P 2))) using
    equivalence" $ do
    let g = Box (con (P 1) (Neg (P 2)))
        klForm = translateFormToKL g
        selForm = fromJust (translateFormToKr klForm)
    areEquivalent smallModels g selForm `shouldBe` True


it "translation succeeds for S5 model (exampleModel3)" $ do
    isJust (kripkeToKL exampleModel3 (makeWorldState 30)) `shouldBe` True

it "translation succeeds for clustered model (exampleModel6)" $ do
    isJust (kripkeToKL exampleModel6 (makeWorldState 60)) `shouldBe` True
```

```haskell
            it "translation fails for linear non-transitive model (exampleModel2)" $ do
                isNothing (kripkeToKL exampleModel2 (makeWorldState 20)) `shouldBe` True

            it "translation fails for cyclic non-transitive model (exampleModel5)" $ do
                isNothing (kripkeToKL exampleModel5 (makeWorldState 50)) `shouldBe` True

            it "preserves truth for atomic proposition P 1 in S5 model (exampleModel3)" $ do
                testTruthPres exampleModel3 (makeWorldState 30) (P 1) `shouldBe` Just True

            it "preserves truth for box operator Box (P 1) in S5 model (exampleModel3)" $ do
                testTruthPres exampleModel3 (makeWorldState 30) (Box (P 1)) `shouldBe` Just True

            it "preserves truth for diamond operator dia (P 1) in S5 model (exampleModel3)" $ do
                testTruthPres exampleModel3 (makeWorldState 30) (dia (P 1)) `shouldBe` Just True

            it "preserves truth for conjunction con (P 1) (P 2) in clustered model (exampleModel6)" $ do
                testTruthPres exampleModel6 (makeWorldState 60) (con (P 1) (P 2)) `shouldBe` Just True

            it "verifies contradiction is false in modelKL7" $ do
                not (checkModel modelKL7 (translateFormToKL (con (P 2) (Neg (P 2))))) `shouldBe` True

            it "verifies reflexivity axiom true in modelKL7b (w71 in cluster)" $ do
                checkModel modelKL7b ref `shouldBe` True

            it "always creates an empty domain" $ do
                 domain (fromJust (kripkeToKL exampleModel3 (makeWorldState 30))) `shouldBe` Set.empty

areEquivalent :: (Eq a, Ord a) => [KripkeModel a] -> ModForm -> ModForm -> Bool
areEquivalent models f1 f2 =
    all (\m -> all (\w -> makesTrue (m, w) f1 == makesTrue (m, w) f2) (universe m)) models

testTruthPres :: KripkeModel WorldState -> WorldState -> ModForm -> Maybe Bool
testTruthPres km w g =
    case kripkeToKL km w of
    Nothing -> Nothing
    Just klModel -> Just $ makesTrue (km, w) g == satisfiesModel klModel (translateFormToKL g)

dia :: ModForm -> ModForm
dia f = Neg (Box (Neg f))


(<==>) :: Bool -> Bool -> Bool
(<==>) p q = (p && q) || (not p && not q)
```

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.