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

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

Sunday 30$^{\text{th}}$ 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.

Symbols of $\mathcal{KL}$
The language $\mathcal{KL}$ is built on the following alphabet:

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

In 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.

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

-- Represents a first-order variable (e.g., "x")
newtype Variable = Var String deriving (Eq, Ord, Show)
```

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. 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), we also define `PrimitiveTerm` and `PrimitiveAtom`. These definitions will be used for defining the semantics later.

```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)

-- 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)

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

-- KL-formulas
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)
```

```haskell
-- 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 $\mathcal{KL}$ introduces a knowledge operator **K** and uses an infinite domain $\mathcal{N}$ of standard names to denote individuals. $\mathcal{KL}$ is designed to model knowledge and uncertainty, as detailed in Levesque and Lakemeyer (2001).
Formulas of $\mathcal{KL}$ are evaluated in world states, which are consistent valuations of atoms and terms, while epistemic states consist of multiple possible worlds, reflecting epistemic possibilities. The semantics is 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.

```
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
```

Worlds and Epistemic States

A `WorldState` represents a single possible world in $\mathcal{KL}$, mapping primitive atoms to truth values and primitive terms to standard names. We implemented it as mapping from atoms and terms instead of just primitive ones; but we make sure to only ever actually use *primitive* atoms and *primitive* terms when creating a `WorldState` (using 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)

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

Constructing World States

We can construct world states using `mkWorldState`, which builds a `WorldState` from lists of primitive atoms and terms. 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, mathematically speaking, are atoms and terms as well), `mkWorldState` first converts the primitive constructors to those of regular terms and atoms. It then uses 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 ensuring we abide by the single-valuation principle (Levesque and Lakemeyer 2001, p. 24). Finally, `mkWorldState` constructs maps for efficient lookup.

```
-- Constructs a WorldState from primitive atoms and primitive terms
mkWorldState :: [(PrimitiveAtom, Bool)] -> [(PrimitiveTerm, StdName)] -> WorldState
```

```haskell
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))

-- 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 decided to have `mkWorldState` change the constructors of data type `PrimitiveAtom` or `PrimitiveTerm` to those of `Atom` and `Term`, we also implemented two helper functions to check whether a `Term` or an `Atom` is primitive.

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

```haskell
-- Check whether 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 whether a formula is true in a model, we define a type for $\mathcal{KL}$ models:

```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)
```

A `Model` consists of an actual world, an epistemic state, and a domain. The function `satisfiesModel` implements $\mathcal{KL}$'s satisfaction relation.

```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 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 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 chose is computationally expensive. However, we chose because, (i), it is more flexible and allows for more varied usage, and, (ii), it 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
```

```haskell
      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

A knowledge base is a collection of symbolic structures representing an agent's beliefs, and idealy, we can then reason using these encoded beliefs. 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.

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.

```haskell
-- 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.

```haskell
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.

```haskell
-- 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.

```haskell
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.

```haskell
-- 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

We want to compare $\mathcal{KL}$ and Propositional Modal Logic based on Kripke frames (denoted 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 since epistemic states act as equivalence classes on the accessibility relation.

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.

```
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)
```

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

```
--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:

```
-- 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
  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:

```
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`:

```
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.

```
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`

We propose that reasonable translation functions should at least satisfy the following 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 Translating 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

The function `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"
```

```haskell
--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

The function `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 $\mathcal{KL}$.

```haskell
-- Translates 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

The function `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. By 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 $V : \mathit{Prop} \to \mathcal{P}(W)$ that assigns to 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 other things, 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 generally 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 ($\forall u, v, w((Ruv \land Rvw) \rightarrow Ruw)$) and euclidean ($\forall u, v, w((Ruv \land Ruw) \rightarrow Rvw)$). For this, we implement the following two functions that check whether a Kripke model is transitive and euclidean, respectively:

```
-- Checks whether 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 whether 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 whether 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: due to time constraints we omitted function symbol evaluation, limiting the satisfiability and validity checking to a propositional-like subset of $\mathcal{KL}$.
This section 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. 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: terminating 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`/ modules.

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).

With $\mathcal{KL}$ 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., n1, n2,...) 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, used parameters, and processed subformula of ancestor nodes. The inclusion of `retainedSubFormula` is important for the tableau expansion process, as it allows us to keep track of the formulas that could be used to close a branch (see `isClosed`) while differentiating from formula that can be further processed (see `applyRule`).

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

data Node = Node Formula TabWorld deriving (Eq, Show)

data Branch = Branch { nodes :: [Node],  params :: Set StdName, retainedSubFormula :: [Node
    ] } deriving (Eq, Show)
```

Tableau Rules
Rules decompose formulas, producing either a closed branch (contradictory) or open branches (consistent). The function`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).

```
data RuleResult = Closed | Open [Branch] deriving (Eq, Show)

-- Generates fresh parameters not in the used set
```

```
newParams :: Set StdName -> [StdName]
newParams used = [StdName ("n" ++ show i) | i <- [(1::Int)..], StdName ("n" ++ show i) `Set
    .notMember` used]

applyRule :: Node -> Branch -> RuleResult
applyRule (Node f w) branch = case f of
  Atom g -> Open [Branch (nodes branch) (params branch) (retainedSubFormula branch ++ [Node
       (Atom g) w])]  -- If formula is an atom: Do nothing; keep the formula in the branch
       .
  Not (Atom g) -> Open [Branch (nodes branch) (params branch) (retainedSubFormula branch ++
       [Node (Not (Atom g)) w])]  -- Negated atoms remain, checked by isClosed
  Equal t1 t2 -> if t1 == t2 then Open [branch] else Closed -- Reflexive equality
  Not (Equal t1 t2) -> if t1 == t2 then Closed else Open [branch] -- Contradiction for t /=
       t
  Not (Not f') -> Open [Branch (Node f' w : nodes branch) (params branch) (
      retainedSubFormula 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) (retainedSubFormula branch)] -- De Morgan: ~(f1 v f2) -> ~f1 & ~f2
  Not (Exists x f') -> Open [Branch (Node (klforall x (Not f')) w : nodes branch) (params
      branch) (retainedSubFormula 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) (retainedSubFormula
      branch)
                   , Branch (Node f2 w : nodes branch) (params branch) (retainedSubFormula
                       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)) (retainedSubFormula branch)
    in Open [newBranch]
  K f' -> Open [expandK f' w branch] -- Knowledge rule, add formula to a new world

expandK :: Formula -> TabWorld -> Branch -> Branch
expandK f _ branch = Branch (Node f 1 : nodes branch) (params branch) (retainedSubFormula
    branch) --- Only world 1

expandKNot :: Formula -> TabWorld -> Branch -> Branch
expandKNot f _ branch = Branch (Node (Not f) 2 : nodes branch) (params branch) (
    retainedSubFormula branch) ---Only world 1
```

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. We acknowledge that the function `isClosed` may have a better implementation. The current implementation is the result of resolving a bug found (using the package `Debug.Trace`) at last minute.

The function works as follows: first, we collect the atoms (`(a, w, True)` for positive atoms `(Node (Atom a) w)`; `(a, w, False)` for negated atoms `(Node (Not (Atom a)) w)`). For example, if `nodes = [Node (Atom P(n1)) 0, Node (Not (Atom P(n1))) 0]`, then `atoms = [(P(n1), 0, True), (P(n1), 0, False)]`. Next, we collect the equalities. After this, we check the atom contradictions. There we use `any` to find pairs in `atoms` and return `True` if a contradiction exists. In a subsequent step, we check for equality contradictions. The result of the function is a disjunction of all the possible types of contradictions that may be found. Clearly the disjunction is `True` if any 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.

```
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]
      retainedAtoms = [(a, w, True) | Node (Atom a) w <- retainedSubFormula b]
              ++ [(a, w, False) | Node (Not (Atom a)) w <- retainedSubFormula 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]
      retainedAtomsContraActual = any (\(a1, w1, b1) -> any (\(a2, w2, b2) ->
                    a1 == a2 && w1 == 0 && w2 == 0 && b1 /= b2) retainedAtoms)
                        retainedAtoms
      retainedAtomsContraModal = any (\(a1, w1, b1) -> any (\(a2, w2, b2) ->
                    a1 == a2 && ((w1 == 1 && w2 == 2) || (w1 == 2 && w2 == 1) || (w1 == 1
                        &&  w2 == 1)) && b1 /= b2) retainedAtoms) retainedAtoms
      atomContraActual = any (\(a1, w1, b1) -> any (\(a2, w2, b2) ->
                    a1 == a2 && w1 == 0 && w2 == 0 && b1 /= b2) atoms) atoms
      atomContraModal = any (\(a1, w1, b1) -> any (\(a2, w2, b2) ->
                    a1 == a2 && ((w1 == 1 && w2 == 2) || (w1 == 2 && w2 == 1) || (w1 == 1
                        && w2 == 1)) && b1 /= b2) atoms) atoms
      eqContraActual = any (\((t1, t2), w1, b1) -> any (\((t3, t4), w2, b2) ->
                    t1 == t3 && t2 == t4 && w1 == 0 && w2 == 0 && b1 /= b2) equals) equals
      eqContraModal = any (\((t1, t2), w1, b1) -> any (\((t3, t4), w2, b2) ->
                    t1 == t3 && t2 == t4 && ((w1 == 1 && w2 == 2) || (w1 == 2 && w2 == 1)
                        || (w1 == 1 && w2 == 1)) && b1 /= b2) equals) equals
  in atomContraActual || atomContraModal || eqContraActual || eqContraModal ||
      retainedAtomsContraActual || retainedAtomsContraModal -- 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.

```
expandTableau :: [Branch] -> Maybe [Branch]
expandTableau branches
  | null branches = Nothing
  | all isClosed branches = Nothing
  | otherwise =
      let openBranches = filter (not . isClosed) branches
          expandable = filter (not . null . nodes) openBranches
      in if null expandable
         then Just openBranches
         else case expandable of
             (branch:rest) ->
                 let ruleResult = applyRule (head (nodes branch)) (Branch (tail (nodes
                     branch)) (params branch) (retainedSubFormula branch))
                 in case ruleResult of
                      Closed -> expandTableau rest
                      Open newBranches -> case expandTableau rest of
                          Nothing -> expandTableau newBranches
                          Just restBranches -> case expandTableau newBranches of
                              Nothing -> Just restBranches
                              Just newBs -> Just (newBs ++ restBranches)
             [] -> Nothing -- This case should never occur due to the earlier null check
```

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.

```haskell
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 $\vee$, $\exists$). In a next step, `isClosed` checks each branch for contradictions, guiding `expandTableau` to prune closed branches or halt with an open one.

```haskell
isValid :: Formula -> Bool
isValid f = not (isSatisfiable (Not f))
```

5 Tests

Testing is done using the Hspec testing framework and QuickCheck for property-based testing. The tests are organized into different modules, each focusing on a specific aspect of the code. The main test file is `Spec.lhs`, which serves as an entry point for running all the tests. This file uses the Hspec framework to automatically discover and run all the tests in the project. The `hspec-discover` tool automatically finds all the test files with the suffix `Spec.lhs` in the `test` directory and runs them.

```haskell
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

This project follows the convention of one spec file per module. Each spec file includes both example and property based tests. The example tests are used to verify the correctness of the code, while the property-based tests are used to check that the code behaves correctly for a wide range of inputs. Test fixtures where used where appropriate to avoid code duplication. Custom generators are used to create random inputs for some of property-based tests. These generators can either be found in the `Test/Generators.lhs` file or as an instance of the `Arbitrary` typeclass alongside the relevant type. Below will be a couple of our custom generators, followed by a few examples of the tests we have written.

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

```haskell
-- 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)
```

```
spec :: Spec
spec =  describe "evalTerm - Example 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 "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
```

```
        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
```

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.