references.bib

My Report

Me

Monday 17$^{\text{th}}$ March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LaTeXeditor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with your code: `stack ghci`
- To run the executable from Section 4: `stack build && stack exec myprogram`
- To run the tests from Section 5: `stack clean && stack test --coverage`

1.1 Syntax of $\mathcal{KL}$

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

```
module SyntaxKL where
```

Symbols of $\mathcal{KL}$
The expressions of $\mathcal{KL}$ are constituted by sequences of symbols drawn from the following two sets (cf. [?]): Firstly, the *logical symbols*, which consist of the logical connectives and quantifiers $\exists, \vee, \neg$, as well as punctuation and parentheses. Furthermore, it compromises a countably infinite supply of first-order variables denoted by the set $\{x, y, z, \ldots\}$, a countably infinite supply of standard names, represented by the set $\{\#1, \#2, \ldots\}$, and the equality symbol $=$. The *non-logical symbols* comprise predicate symbols of any arity $\{P, Q, R, \ldots\}$, which are intended to represent domain-specific properties and relations, and function symbols of any arity, which are used to denote mappings from individuals to individuals ([?], p.22).
n this implementation, standard names are represented as strings (e.g., "n1", "n2") via the StdName type, and variables are similarly encoded as strings (e.g., "x", "y") with the Variable type, ensuring that we have a distinct yet infinite supplies of each.

```
-- 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 and Atoms

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

```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")
          | FuncApp 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"
                   | PFuncApp 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)
```

Formulas

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

```haskell
--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, Show)

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

-- Implication as derived form
implies :: Formula -> Formula -> Formula
implies f1 f2 = Or (Not f1) f2

-- 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 $\mathcal{K}\updownarrow$'s syntax to implement the semantics.

2 $\mathcal{KL}$: Syntax and Semantics

2.1 Semantics of $\mathcal{KL}$

$\mathcal{KL}$ is an epistemic extension of first-order logic designed to model knowledge and uncertainty, as detailed in Lokb. It introduces a knowledge operator K and uses an infinite domain $\mathcal{N}$ of standard names to denote individuals. Formulas are evaluated in world states: consistent valuations of atoms and terms, while epistemic states capture multiple possible worlds, reflecting epistemic possibilities.

The semantics are implemented in the SemanticsKL module, which imports syntactic definitions from SyntaxKL and uses Haskell's Data.Map and Data.Set for efficient and consistent mappings.

```
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 truth values to primitive atoms and standard names to primitive terms. An EpistemicState, defined as a set of WorldStates, models the set of worlds an agent considers possible, enabling the evaluation of the K operator.

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

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

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

```
-- Constructs a WorldState from primitive atoms and primitive terms
mkWorldState :: [(PrimitiveAtom, Bool)] -> [(PrimitiveTerm, StdName)] -> WorldState
mkWorldState atoms terms =
  let convertAtom (PPred p ns, b) = (Pred p (map StdNameTerm ns), b)  -- Convert primitive
      atom to Atom
      convertTerm (PStdNameTerm n, v) = (StdNameTerm n, v)  -- Convert primitive term to
          Term
      convertTerm (PFuncApp f ns, v) = (FuncApp f (map StdNameTerm ns), v)
      atomList = map convertAtom atoms
      termList = map convertTerm terms
  in WorldState (Map.fromList (checkDups atomList)) (Map.fromList (checkDups termList))

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

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

```
-- Checks if a term is primitive (contains only standard names)
```

```
isPrimitiveTerm :: Term -> Bool
isPrimitiveTerm (StdNameTerm _) = True
isPrimitiveTerm (FuncApp _ args) = all isStdName args
  where isStdName (StdNameTerm _) = True
        isStdName _ = False
isPrimitiveTerm _ = False

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

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

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

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

3. FuncApp f args
   If the term is a function application (e.g., f(n1,n2)), evalTerm evaluates the argument, by recursively computing the StdName values of each argument in args using evalTerm w. Next, the ground term is constructed: It Builds a new FuncApp term where all arguments are standard names (wrapped in StdNameTerm), ensuring it's fully ground. We then look up the value by querying the termValues map in the world state w for the denotation of this ground term, erroring on undefined terms.

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

Groundness and Substitution

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

```
-- Check if a term is ground (contains no variables).
isGround :: Term -> Bool
isGround t = case t of
  VarTerm _ -> False
  StdNameTerm _ -> True
  FuncApp _ 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
  FuncApp f args -> FuncApp f (map (substTerm x n) args)

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

Model and Satisfiability

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

```
-- Represents a model with an actual world, epistemic state, and domain
data Model = Model
  { actualWorld :: WorldState       -- The actual world state
  , epistemicState :: EpistemicState -- Set of possible world states
  , domain :: Set StdName           -- Domain of standard names
  } deriving (Show)
```

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

```
-- Checks if a formula is satisfied in a model
satisfiesModel :: Model -> Formula -> Bool
satisfiesModel m = satisfies (epistemicState m) (actualWorld m)
  where
    satisfies e w formula = case formula of
      Atom (Pred p terms) ->
        if all isGround terms
          then Map.findWithDefault False (Pred p terms) (atomValues w)   -- Default False
              for undefined atoms
          else error "Non-ground atom in satisfies!"
      Equal t1 t2 ->
        if isGround t1 && isGround t2   -- Equality of denotations
          then evalTerm w t1 == evalTerm w t2
          else error "Non-ground equality in satisfies!"
      Not f ->
        not (satisfies e w f)
      Or f1 f2 ->
```

```
        satisfies e w f1 || satisfies e w f2
      Exists x f ->
      -- \(e, w \models \exists x. \alpha\) iff for some name \(n\), \(e, w \models \
          alpha_n^x\)
        any (\n -> satisfies e w (subst x n f)) (Set.toList $ domain m)
      -- \(e, w \models K \alpha\) iff for every \(w' \in e\), \(e, w' \models \alpha\)
      K f ->
        all (\w' -> satisfies e w' f) e
```

Grounding and Model Checking

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

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

Note that we use the function groundFormula here. Since we have implemented satisfiesModel such that it assumes ground formulas or errors out, we decided to handle free variables by grounding formulas by substituting free variables. We implement groundFormula as follows:

```
-- Generates all ground instances of a formula
groundFormula :: Formula -> Set StdName -> [Formula]
groundFormula f dom = do
  -- converts the set of free variables in f to a list
  let fvs = Set.toList (freeVars f)
  -- creates a list of all possible assignments of domain elements to each free variable
  -- For each variable in fvs, toList domain provides the list of standard names, mapM
      applies this (monadically), producing all the combinations
  subs <- mapM (\_ -> Set.toList dom) fvs
  --iteratively substitute each variable v with a standard name n in the formula
  return $ foldl (\acc (v, n) -> subst v n acc) f (zip fvs subs)
```

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 freeVars that identifies all the variables in a formula that need grounding or substitution. It takes a formula and returns a Set Variable containing all the free variables in that formula:

```
-- Collects free variables in a formula
freeVars :: Formula -> Set Variable
freeVars f = case f of
  --Collects free variables from all terms in the predicate
  Atom (Pred _ terms) -> Set.unions (map freeVarsTerm terms)
  -- Unions the free variables from both terms t1 and t2.
  Equal t1 t2 -> freeVarsTerm t1 `Set.union` freeVarsTerm t2
  -- Recursively computes free variables in the negated subformula f'.
  Not f' -> freeVars f'
  -- Unions the free variables from both disjuncts f1 and f2.
  Or f1 f2 -> freeVars f1 `Set.union` freeVars f2
  -- Computes free variables in f', then removes x (the bound variable) using delete, since
       x is not free within \exists x f'
  Exists x f' -> Set.delete x (freeVars f')
  -- Recursively computes free variables in f', as the K operator doesn't bind variables.
  K f' -> freeVars f'
  where
    freeVarsTerm t = case t of
      --A variable (e.g., x) leads to a singleton set containing v.
      VarTerm v -> Set.singleton v
      -- A standard name (e.g., n1) has no free variables, so returns an empty set.
      StdNameTerm _ -> Set.empty
```

```
    -- A function application (e.g., f(x,n1)) recursively computes free variables in its
        arguments.
    FuncApp _ args -> Set.unions (map freeVarsTerm args)
```

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

Note: For the Beta-version, we omitted function symbol evaluation, limiting the satisfiability and validity checking to a propositional-like subset.

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

```
module Tableau where

import SyntaxKL
import SemanticsKL
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Map (Map)          -- Added for WorldState maps
import qualified Data.Map as Map
```

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 ([?], p.23). The tableau method handles this via parameters (free variables) and δ-rules (existential instantiation), introducing new names as needed. This means that we Use a countably infinite supply of parameters (e.g., a1,a2,...) instead of enumerating all standard names.
- Modal handling: The K-operator requires branching over possible worlds within an epistemic state.

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

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

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

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

Tableau Rules

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

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

-- Generates fresh parameters not in the used set
newParams :: Set StdName -> [StdName]
newParams used = [StdName ("a" ++ show i) | i <- [1..], 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.
  Equal t1 t2 -> Open [branch]  -- Keep equality as is; closure checks congruence
  Not (Equal t1 t2) -> 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 (for_all 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 f w branch = Branch (Node f (1 + maxWorld) : nodes branch) (params branch)
  where maxWorld = maximum (0 : [w' | Node _ w' <- nodes branch])

-- Expands \not K \varphi to a new world
expandKNot f w branch = Branch (Node (Not f) (1 + maxWorld) : nodes branch) (params branch)
  where maxWorld = maximum (0 : [w' | Node _ w' <- nodes branch])
```

Branch Clodure

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

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

Tableau Expasion

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

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

Top-Level Checkers

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

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

The three function isSatisfiable, expandTableau, and isClosed interact as follows: isSatisfiable starts the process with a single branch containing the formula. expandTableau recursively applies applyRule to decompose formulas, creating new branches as needed (e.g., for $\vee$, $\exists$). isClosed checks each branch for contradictions, guiding expandTableau to prune closed branches or halt with an open one.

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

4 Wrapping it up in an exectuable

We will now use the library form Section ?? in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

5 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Basics

import Test.Hspec
import Test.QuickCheck
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```haskell
main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test` Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.

6 Conclusion

Finally, we can see that [LW13] is a nice paper.

References

[Knu11] Donald E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley Professional, 2011.

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.