Model Checking BSML

N. Cohen, T. Coroneo, V. Iyer, G. Kaminer, and L. Madison

Sunday 30th March, 2025

Abstract

Bilateral State-based Modal Logic (BSML) was introduced in [?] to model Free-Choice (*FC*) and related inferences which arise from speakers' distaste for interpretations that verify a sentence by empty configuration (*neglect-zero tendency*). To preclude such empty configurations, BSML uses state-based semantics (propositions evaluated at sets of worlds rather than individual worlds) and extends basic modal logic with a pragmatic enrichment function $[]^+$ which requires supporting states to be nonempty. We implement an explicit model checker for $\mathrm{BSML}^{\mathbb{V}}$, an extension of BSML with global disjunction, and use this to verify some properties, like Free-Choice inference, as described in [?]. Further, we implement a type for representing Natural Deduction proofs for $\mathrm{BSML}^{\mathbb{V}}$.

Contents

1 Introduction

This section outlines the motivation for BSML by walking through a simple example from [?]. Readers interested only in the implementation can safely skip this section.

1.1 Motivating example

Free-Choice (FC) inferences are instances of a disjunctive sentence ("or") unexpectedly yielding a conjunctive reading ("and"). In the following example, a modalized disjunction (1) yields a conjunction of modals (2).

1. $\Diamond(b \vee c)$ You may go to the beach **or** to the cinema
2. $\Diamond b \wedge \Diamond c$ You may go to the beach **and** you may go to the cinema

Aloni [?] posits that speakers interpret a sentence by identifying structures of reality that reflect it. Such a structure, or "information state", is a set of possible worlds.

In our disjunction, there are four associated worlds:

1. W_{bc} in which both b and c are true (you go to both the beach and the cinema)
2. W_b in which b is true (you go to the beach)
3. W_c in which c is true (you go to the cinema)
4. W_z in which neither b nor c is true (you don't go anywhere)

We can use BSML to model the information states (i.e. sets of worlds) in which the disjunction $b \vee c$ is assertable (or rejectable). A disjunction $\varphi \vee \psi$ is assertable in a state s if s is the union of two substates t and u, where φ is assertable in t and ψ is assertable in u.

1. Where state s_1 is the union of W_b and W_c, $b \vee c$ is assertable since b is assertable in W_b and c is assertable in W_c
2. Where state s_2 is the union of W_{bc} and W_b (or W_{bc} and W_c), $b \vee c$ is assertable since b is assertable in W_b and c is assertable in W_{bc}
3. Where state s_3 is the union of W_b and the empty set (or W_c and the empty set), $b \vee c$ is assertable since

   since each of the disjuncts is assertable in a substate. b is assertable in W_b, and c is supportable in the empty state.
4. In a state consisting of W_z and W_b (and any other state which includes W_z), $b \vee c$ is *not* assertable because in W_z both b and c are false, so no substate containing W_z would allow assertion of b or c.

So, among the four types of states, $b \vee c$ is assertable in all but the last. This is a problem, because if $b \vee c$ is assertable in a state consisting only of W_b (or a state consisting of only W_c) then we have that

$\Diamond(b \vee c)$ is true while $\Diamond b \wedge \Diamond c$ is false, so the FC inference fails. The problematic state, then, is the zero-model: one of the states which it uses to satisfy the disjunction is the empty state.

How do we account for FC inferences then? Aloni argues pragmatically that a speaker would not consider the zero-model as one of the candidate states. Neglecting the zero model then, the FC inference would hold because the only states that would support $b \vee c$ would be (1) or (2). To model neglect-zero (to make sure that $b \vee c$ is not assertable in the zero-model), we require that to satisfy a disjunction, the state must be the union of two non-empty substates rather than just the union of two substates. This is modelled by enriching formulas using a *pragmatic enrichment function* which conjuncts to each subformula a non-emptiness atom (`NE`), which requires supporting states to be inhabited.

The enrichment of $b \vee c$ (denoted $[b \vee c]^{+}$) is no longer assertable in a state consisting of only W_b (or a state consisting of only W_c) since `NE` would not be assertable in any substates that could (vacuously) support c. Finally then, since the only states in which the enriched disjunction holds are (1) and (2), the FC inference holds.

1.2 Our contribution

This report details our implementation of an explicit model checker for BSML. In the last section, we create a framework for writing Natural Deduction style proofs for BSML in Haskell.

Developing model-checkers is useful for any logic - it helps us understand quickly whether a formula is true on a particular model (while removing any human error involved in this process), and also makes it easy for the user to check the validity of a class of formulas on a class of models quickly. The goal is to ensure model-checkers are accurate and efficient: while in BSML the latter criterion is hard to achieve on account of the unique semantics of the logic, we have tried to ensure that the model-checker is as efficient as possible while remaining sound.

Our implementation for Natural Deduction can be used as a (quite primitive for now) interactive theorem prover for BSML. Representing these proofs is also the first step to potentially building a functional automated theorem prover for the language. As with model-checkers, this makes the job of anyone working in the field easier as they may now systematically verify the correctness of their proofs. Our representation of Natural Deduction is built by creating a new `Proof` type, and ensuring that the only way one can get from one proof to another is by following the axiomatization of BSML outlined in [**?**].

2 Model checking BSML

This section describes the implementation of the explicit model checker for BSML. More precisely, we will focus on BSML$^{\mathbb{V}}$, an extension of BSML with so-called *global disjunction*. This extension was not chosen for conceptual reasons, but merely to make the implementation of Natural Deduction more palatable, as we will see later. Throughout this report, we will not need to differentiate between BSML and BSML$^{\mathbb{V}}$, so we will be somewhat sloppy and simply write BSML for our language.

2.1 Syntax

```
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE RankNTypes #-}
module Syntax where
```

```
import Data.Data (Data)

import Control.Lens
import Control.Lens.Extras (is)

import Test.QuickCheck
```

This module describes the syntactical elements of BSML. We define formulas, some syntactical operations and provide generators for random formulas.

The formulas of BSML are defined as follows:

```
type Proposition = Int

-- Formulas of BSML.
data Form
  = Bot
  | NE
  | Prop Proposition
  | Neg Form
  | And Form Form
  | Or  Form Form
  | Gor Form Form
  | Dia Form
  deriving (Eq,Ord,Show,Data)
```

Readers familiar with Modal Logic (see e.g.[?]) should recognize this as the basic modal language, extended with NE, the nonemptiness atom, and $\mathbin{\mathbb{V}}$, global disjunction. As we will see when defining the semantics, NEis used to exclude the assertion of logical statements due to empty information-configurations (i.e. empty states/teams).

For the sake of more legible output, we also define a pretty-printer for formulas. Note that we print $\mathbin{\mathbb{V}}$ as V/ rather than \V to avoid having to worry about the escape character.

```
ppForm :: Form -> String
ppForm = \case
  Bot       -> "_|_"
  NE        -> "NE"
  Prop p    -> show p
  Neg f     -> "~" ++ ppForm f
  And f1 f2 -> "(" ++ ppForm f1 ++ " & " ++ ppForm f2 ++ ")"
  Or f1 f2  -> "(" ++ ppForm f1 ++ " v " ++ ppForm f2 ++ ")"
  Gor f1 f2 -> "(" ++ ppForm f1 ++ " V/ " ++ ppForm f2 ++ ")"
  Dia f     -> "<>" ++ ppForm f
```

Further, we define some abbreviations for formulas, following [?]. As usual, $\Box$ is defined as the dual of $\Diamond$. As will become evident when we define the semantics, $\bot$ is supported in a state if and only if that state is empty. It is therefore referred to as the *weak contradiction*. The strong contradiction, defined as $\mathbb{\bot} := \bot \land \mathtt{NE}$, will never be supported. Dually, the formula $\top :=$ NE serves as the *weak tautology*, being supported by non-empty states and the `strong tautology` $\mathbb{\top} := \neg\bot$ is always supported.

```
-- Define box as the dual of diamond.
box :: Form -> Form
box = Neg . Dia . Neg

-- Define the strong contradiction (which is never assertable).
botbot :: Form
botbot = And Bot NE

-- NE functions as a weak tautology (assertable in non-empty states).
top :: Form
top = NE

-- Define the strong tautology (which is always assertable).
toptop :: Form
```

```
toptop = Neg Bot
```

As in [?], we can use these notions of contradiction and tautology to interpret finite (global) disjunctions and conjunctions:

```
bigOr :: [Form] -> Form
bigOr [] = Bot
bigOr fs = foldr1 Or fs

bigAnd :: [Form] -> Form
bigAnd [] = toptop
bigAnd fs = foldr1 And fs

bigGor :: [Form] -> Form
bigGor [] = botbot
bigGor fs = foldr1 Gor fs
```

Note that we could have (semantically equivalently) defined e.g.

```
bigor :: [Form] -> Form
bigor = foldr Or Bot
```

but this would have had the undesired side-effect of including Bot in *every* disjunction, including non-empty ones.

2.1.1 Random formulas

In order to verify some properties of BSML, we would like to be able to generate random formulas. We will use QuickCheck's ecosystem for this purpose, so we only need to define an instance of `Arbitrary` for `Form`.

First, we fix a number of propositions, which we will also use when generating random valuations for models. This guarantees that random models and formulas have a more meaningful interaction, in the sense that the formulas will actually refer to the propositions that occur in the model.

```
-- We use proposition in the range (1, numProps).
numProps :: Int
numProps = 32
```

One might wonder why we do not use the size parameter of a generator to determine the range of propositions. We intentionally avoid this, because it would introduce a bias in the occurence of Propositions, where more nested subformulas will not contain high propositions.

Now we can define the `Arbitrary Form`-instance using a standard sized generator, where formulas generated with size 0 are random atoms and larger formulas are generated by applying a random constructor to random smaller formulas.

```
-- Generate a random atom.
randomAtom :: Gen Form
randomAtom = oneof [Prop <$> choose (1, numProps), pure NE, pure Bot]

instance Arbitrary Form where
  arbitrary = sized $ \case
    0 -> randomAtom
    _ ->  oneof [
        randomAtom,
        Neg <$> f,
        And <$> f <*> f,
        Or  <$> f <*> f,
        Gor <$> f <*> f,
        Dia <$> f
```

```
  ]
  where f = scale (`div` 2) arbitrary
```

The choice to scale the size of the generator by dividing it by 2 is completely arbitrary, but seems to work well in practice and is used in similar projects, see e.g. [?].

Last, we also define shrinks of formulas that empower QuickCheck to attempt simplifying counterexamples when/if it finds any.

```
shrink (Neg f)     = [Bot, NE, f]       ++ [Neg f'       | f'        <- shrink f]
shrink (Dia f)     = [Bot, NE, f]       ++ [Dia f'       | f'        <- shrink f]
shrink (And f1 f2) = [Bot, NE, f1, f2] ++ [And f1' f2' | (f1',f2') <- shrink (f1,f2)]
shrink (Or f1 f2)  = [Bot, NE, f1, f2] ++ [Or  f1' f2' | (f1',f2') <- shrink (f1,f2)]
shrink (Gor f1 f2) = [Bot, NE, f1, f2] ++ [Gor f1' f2' | (f1',f2') <- shrink (f1,f2)]
shrink _           = []
```

2.1.2 Boilerplate for subformulas

This section is slightly technical and can safely be skipped. It introduces some functions that allow us to check properties of subformulas (e.g. whether a formula contains `NE` anywhere in its subformulas).

The `Lens`-library defines a typeclass `Plated` that implements a lot of boilerplate code for the transitive descendants of values of recursively defined types, so let us make `Form` an instance.

```
-- Use default implementation: plate = uniplate
instance Plated Form
```

We also derive a prism corresponding to every constructor of `Form`.

```
-- Derives prisms _Bot, _NE, ..., _Gor, _Dia
makePrisms ''Form
```

For readers unfamiliar with this, a`Prism` is to a constructors what a `Lens` is to a field. For example, the derived `Prism` for `Or` is of type

```
_Or :: Prism' Form (Form, Form)
```

which can loosely be interpreted as a pair of functions; one that turns a `(Form, Form)` into a `Form` (by applying `Or` in this case), and one that tries to turn a `Form` into a `(Form, Form)` (in this case, by taking the arguments out of the constructor if the formula is a disjunction). As we are familiar with from `Lens`, this is suitably generalized.

Now, we can e.g. check whether a formula uses the constructors `NE` or `Gor` by seeing if any of its transitive descendants is built using one of these constructors.

```
isBasic :: Form -> Bool
isBasic = any ((||) . is _NE <*> is _Gor) . universe
```

Or more generally, check whether a certain constructor was used.

```
hasCr :: Prism' Form fs -> Form -> Bool
hasCr = (. universe) . any . is
```

2.2 Semantics

We quickly recall the most important aspects of the semantics of BSML from [?]. We interpret formulas on (Kripke) models, which consist of

- a set of worlds W;
- a binary relation $R \subseteq W \times W$ between worlds;
- and a valuation $V : W \to \wp(\texttt{Prop})$ mapping a world to the propositions that hold in it. [1]

A *team* or *state* (we will use these terms interchangably) on a model is a subset $s \subseteq W$. To link back to the introduction, the worlds represent information-configurations and a team represents the worlds that a speaker perceives as possible.

As the name *Bilateral State-based Modal Logic* suggests, formulas are evaluated with respect to a team (rather than a world). The "bilateral" part refers to the fact that we make use of *two* fundamental semantics notions; *support* and *anti-support*, rather than just *truth*. Support (resp. antisupport) of a formula by a team represents the speaker's ability to assert (resp. reject) the formula, given the worlds deemed possible in the team.

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
module Semantics where

import Syntax

import Data.List
import Data.Set (Set)
import Data.IntMap (IntMap)
import qualified Data.Set as Set
import qualified Data.IntMap as IntMap

import Test.QuickCheck
```

In our implementation, we will represent worlds as `Int`'s and teams as lists of worlds. For the relation and valuation, we use `IntMap`'s, which are conceptually identical to association lists with `Int`-valued keys, but allow for fast lookup of the successors and satisfied proposition letters of a world.

```haskell
type World = Int
type Team = [World]
type Rel = IntMap Team
type Val = IntMap (Set Proposition)

data KrM = KrM {worlds :: Team,
                rel    :: Rel,
                val    :: Val}
  deriving (Show)
```

We also define the following shorthands for looking up successors and valuations of worlds, that allow us to treat `rel` and `val` as (partial) functions:

```haskell
rel' :: KrM -> World -> Team
rel' = (IntMap.!) . rel

val' :: KrM -> World -> Set Proposition
val' = (IntMap.!) . val
```

[1] In the paper, the valuation is defined as a function $\texttt{Prop} \to \wp W$, but this is easily seen to be equivalent.

To allow defining a (anti)support-relation for different structures as well, we use the following typeclasses, that contain a both a curried and uncurried version of the function for (anti)support for ease of use (only one is required to be provided; the other is the curried/uncurried equivalent).

```
class Supportable model state formula where
  support :: model -> state -> formula -> Bool
  support = curry (|=)

  (|=) :: (model, state) -> formula -> Bool
  (|=) = uncurry support

  {-# MINIMAL (|=) | support #-}

class Antisupportable model state formula where
  antisupport :: model -> state -> formula -> Bool
  antisupport = curry (=|)

  (=|) :: (model, state) -> formula -> Bool
  (=|) = uncurry antisupport

  {-# MINIMAL (=|) | antisupport #-}
```

Before we implement the semantics, we need a function that computes all pairs of teams whose union is a given team s. Naively, we may define e.g.

```
teamParts :: Team -> [(Team, Team)]
teamParts s = [(t,u) | t <- ps, u <- ps, sort . nub (t ++ u) == sort s]
  where ps = subsequences s
```

Computationally however, this is incredibly expensive and will form a major bottleneck for the efficiency of the model checking. While finding such partitions is inherently exponential in complexity, we can still do better (at least on average) than the above:

```
teamParts :: Team -> [(Team, Team)]
teamParts s = do
    t <- subsequences s
    u <- subsequences t
    return (t, s \\ u)
```

Now, we are ready to define our semantics, in accordance to [**?**]:

```
instance Supportable KrM Team Form where
  (_,s) |= Bot     = null s
  (_,s) |= NE      = not (null s)
  (m,s) |= Prop n  = all (elem n . val' m) s
  (m,s) |= Neg f   = (m,s) =| f
  (m,s) |= And f g = (m,s) |= f && (m,s) |= g
  (m,s) |= Or f g  = any (\(t,u) -> (m,t) |= f && (m,u) |= g) $ teamParts s
  (m,s) |= Gor f g = (m,s) |= f || (m,s) |= g
  (m,s) |= Dia f   = all (any (\t -> not (null t) && (m,t) |= f) . subsequences . rel' m) s

instance Antisupportable KrM Team Form where
  _     =| Bot     = True
  (_,s) =| NE      = null s
  (m,s) =| Prop n  = not $ any (elem n . val' m) s
  (m,s) =| Neg f   = (m,s) |= f
  (m,s) =| And f g = any (\(t,u) -> (m,t) =| f && (m,u) =| g) $ teamParts s
  (m,s) =| Or f g  = (m,s) =| f && (m,s) =| g
  (m,s) =| Gor f g = (m,s) =| Or f g
  (m,s) =| Dia f   = all (\w -> (m, rel' m w) =| f) s
```

One may also easily extend the above semantics to lists of formulae, as shown below.

```
instance Supportable KrM Team [Form] where
  support = (all .) . support

instance Antisupportable KrM Team [Form] where
  antisupport = (all .) . antisupport
```

2.3 Random models

As for formulas, we want to be able to generate random models to verify properties of BSML, and will use QuickCheck for this.

We will need a function that generates a `Set` containing random elements of a list, so we define the following analogue of `sublistOf`:

```
subsetOf :: Ord a => [a] -> Gen (Set a)
subsetOf = (Set.fromList <$>) . sublistOf
```

When shrinking the valuation of models, we want to have QuickCheck try valuation with fewer propositions occurring in the model, but it does not make sense to shrink the values of the propositions, so we define the following functions to only shrink which propositions occur:

```
-- Shrink a list without shrinking individual elements
shrinkList' :: [a] -> [[a]]
shrinkList' = shrinkList (const [])

-- Shrink a set without shrinking individual values
shrinkSet :: Set Proposition -> [Set Proposition]
shrinkSet = fmap Set.fromList . shrinkList' . Set.toList

-- Shrink a valuation by uniformly restricting the occuring propositions
shrinkVal :: Val -> [Val]
shrinkVal v = do
  let allProps = Set.unions v
  newProps <- shrinkSet allProps
  return (Set.intersection newProps <$> v)
```

Then, an arbitrary model $M = (W, R, V)$ can be generated as follows:

```
instance Arbitrary KrM where
  arbitrary = sized $ \n -> do
```

The size parameter n gives an upper bound to the amount of worlds; we pick some $k \leq n$ and define $W = 0, 1, \ldots, k$. For every $w <= k$, we then generate a random set of successors $R[w]$ and random set of propositions $V(w)$ that hold at w.

```
    k <- choose (0, n)
    let ws = [0..k]
    r <- IntMap.fromList . zip ws <$> vectorOf (k+1) (sublistOf [0..k])
    v <- IntMap.fromList . zip ws <$> vectorOf (k+1) (subsetOf [1..numProps])
    return (KrM ws r v)
```

When finding counterexamples, it is useful to find models that are as small as possible, so we also define `shrink` that tries to restrict the worlds of the model.

```
  shrink m = do
    ws' <- init $ subsequences $ worlds m
    let r' = IntMap.fromList [(w, rel' m w `intersect` ws') | w <- ws']
    let v' = IntMap.filterWithKey (const . (`elem` ws')) $ val m
    KrM ws' r' <$> shrinkVal v'
```

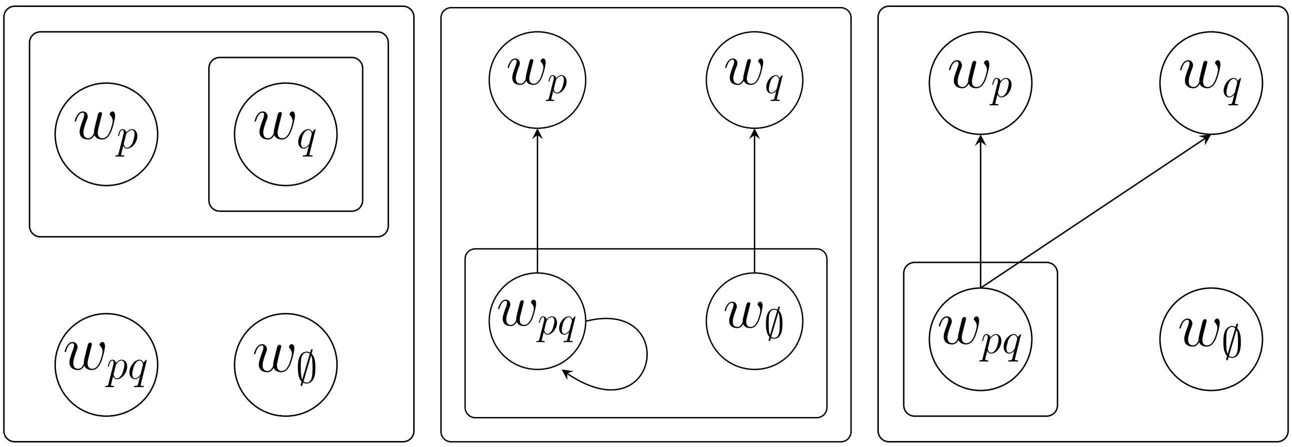
When testing, we will often want to generate a random model *with* a random team or world of that model. Generating a random `Int` or `[Int]` would not work then, since there is no guarantee that the generated value is a valid team/world in the model. To remedy that, we define wrappers for models with a team/world and define `Arbitrary`-instances for those wrappers:

```
data TeamPointedModel = TPM KrM Team
  deriving (Show)

data WorldPointedModel = WPM KrM World
```

```haskell
  deriving (Show)

instance Arbitrary TeamPointedModel where
  arbitrary = do
    m <- arbitrary
    s <- sublistOf $ worlds m
    return (TPM m s)

  shrink (TPM m s) = filter (\(TPM m' s') -> s' `isSubsequenceOf` worlds m') (TPM <$>
      shrink m <*> shrinkList' s)

instance Arbitrary WorldPointedModel where
  arbitrary = do
    m <- arbitrary
    w <- elements $ worlds m
    return (WPM m w)

  shrink (WPM m w) = filter (\(WPM m' w') -> w' `elem` worlds m') (WPM <$> shrink m <*> [w
      ])
```

Note that when shrinking, we should only allow shrinks where the world/team is still contained in the model.

2.4 Verifying basic properties



(a) Example 3a (b) Example 3b (c) Example 3c

Figure 1

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

The following uses the HSpec library to define different tests. We use a mix of QuickCheck and specific inputs, depending on what we are testing for.

The "Figure 3" section corresponds to the three examples labeled 3a, 3b, and 3c [?]. The paper gives a couple formulas per example to illustrate the semantics of BSML. We test each of these formulas to confirm our implementation contains the expected semantics.

The "Figure 3" section corresponds to the three examples labeled 3a, 3b, and 3c in Figure 1 and in the paper [?]. The paper gives a couple formulas per example to illustrate the semantics of BSML. We test each of these formulas to confirm our implementation contains the expected semantics.

```haskell
main :: IO ()
main = hspec $ do
  describe "Figure 3" $ do
```

```
    it "Figure 3a1, p v q" $
      (m3a, s3a1) |= (p `Or` q) `shouldBe` True
    it "Figure 3a1, (p ^ NE) v (q ^ NE)" $
      (m3a, s3a1) |= (And p NE `Or` And q NE) `shouldBe` False
    it "Figure 3a2, (p ^ NE) v (q ^ NE)" $
      (m3a, s3a2) |= (And p NE `Or` And q NE) `shouldBe` True
    it "Figure 3b, <>q" $
      (m3b, s3b) |= Dia q `shouldBe` True
    it "Figure 3b, <>p" $
      (m3b, s3b) |= Dia p `shouldBe` False
    it "Figure 3b, []q" $
      (m3b, s3b) |= box q `shouldBe` False
    it "Figure 3b, []p v []q" $
      (m3b, s3b) |= (box p `Or` box q) `shouldBe` True
    it "Figure 3b, <>p ^ <>q" $
      (m3b, s3b) |= (Dia p `And` Dia q) `shouldBe` False
    it "Figure 3c, <>(p v q)" $
      (m3c, s3c) |= Dia (p `Or` q) `shouldBe` True
```

Below we test the tautologies that should hold for BSML logic ensuring our implementation is correct. Here we use QuickCheck, but we need to limit the maximal size of the arbitrary models we generate. This is necessary because the evaluation of support in team semantics is inherently exponential in complexity (see e.g. the clause for support of disjunctions).

```
  describe "Tautologies" $ modifyMaxSize (const 20) $ do
    modifyMaxSize (const 10) $ prop "box f <==> !<>!f" $
      \(TPM m s) f -> (m,s) |= box (f::Form) == (m,s) |= Neg(Dia (Neg f))

    prop "<>(p v q) <==> <>p v <>q" $
      \(TPM m s) -> (m,s) |= Dia (p `Or` q) == (m,s) |= (Dia p `Or` Dia q)
    prop "<>(p ^ q) ==> <>p ^ <>q" $
      \(TPM m s) -> (m,s) |= Dia (p `And` q) ==> (m,s) |= (Dia p `And` Dia q)
    modifyMaxSize (const 25) $ prop "<>p ^ <>q !==> <>(p ^ q)" $
      expectFailure $ \(TPM m s) -> (m,s) |= (Dia p `And` Dia q) ==> (m,s) |= Dia (p `And`
          q)

    prop "box(p ^ q) <==> box p ^ box q" $
      \(TPM m s) -> (m,s) |= box (p `And` q) == (m,s) |= (box p `And` box q)
    prop "box p v box q ==> box(p v q)" $
      \(TPM m s) -> (m,s) |= (box p `Or` box q) ==> (m,s) |= box (p `Or` q)
    modifyMaxSize (const 25) $ modifyMaxSuccess (const 1000) $ prop "box(p v q) !==> box p
        v box q" $
      expectFailure $ \(TPM m s) -> (m,s) |= box (p `Or` q) ==> (m,s) |= (box p `Or` box q)

    prop "DeMorgan's Law" $
      \(TPM m s) -> (m,s) |= Neg (p `And` q)== (m,s) |= (Neg p `Or` Neg q)

    prop "Dual-Prohibition, !<>(a v b) |= !<>a ^ !<>b" $
      \(TPM m s) -> (m, s) |= Neg (Dia (p `Or` q)) == (m,s) |= (Neg(Dia p) `And` Neg(Dia q)
         )
    prop "strong tautology is always supported" $
      \(TPM m s) -> (m,s) |= toptop
    prop "strong contradiction is never supported" $
      \(TPM m s) -> not $ (m,s) |= botbot
    modifyMaxSize (const 10) $ prop "p v ~p is never supported"  $
      \(TPM m s) -> (m,s) |= (p `Or` Neg p)
    prop "NE v ~NE *can* be supported" $
      expectFailure $ \(TPM m s) -> not $ (m,s) |= (top `Or` Neg top)
    prop "strong tautology !== top" $
      expectFailure $ \(TPM m s) -> (m,s) |= toptop == (m,s) |= top
    where
      p = Prop 1
      q = Prop 2
```

3 Verifying inference

3.1 Pragmatic enrichment

To define the *pragmatic enrichment* function mentioned in the introduction, we define a type to represent formulas of basic Modal Logic (ML) and implement the standard Kripke semantics.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}
module ML where

import Syntax
import Semantics

import Test.QuickCheck

-- Basic Modal Logic formulas
data MForm
  = MProp Proposition
  | MNeg MForm
  | MAnd MForm MForm
  | MOr  MForm MForm
  | MDia MForm
  deriving (Eq,Show)

-- Kripke semantics
instance Supportable KrM World MForm where
  (m,w) |= MProp n  = n `elem` val' m w
  (m,w) |= MNeg f   = not $ (m,w) |= f
  (m,w) |= MAnd f g = (m,w) |= f && (m,w) |= g
  (m,w) |= MOr f g  = (m,w) |= f || (m,w) |= g
  (m,w) |= MDia f   = any (\v -> (m,v) |= f) $ rel' m w
```

Formally, pragmatic enrichment is given by the function $[\cdot]^+ \colon \mathrm{ML} \to \mathrm{BSML}$, defined as

$$
\begin{aligned}
[p]^+ &\coloneqq p \wedge \mathtt{NE} && \text{for } p \in \mathtt{Prop} \\
[\heartsuit\varphi]^+ &\coloneqq \heartsuit[\varphi]^+ \wedge \mathtt{NE} && \text{for } \heartsuit \in \{\neg, \Diamond\} \\
[\varphi \odot \psi]^+ &\coloneqq ([\varphi]^+ \odot [\psi]^+) \wedge \mathtt{NE} && \text{for } \odot \in \{\vee, \wedge\}
\end{aligned}
$$

which is straightforward to implement in Haskell.

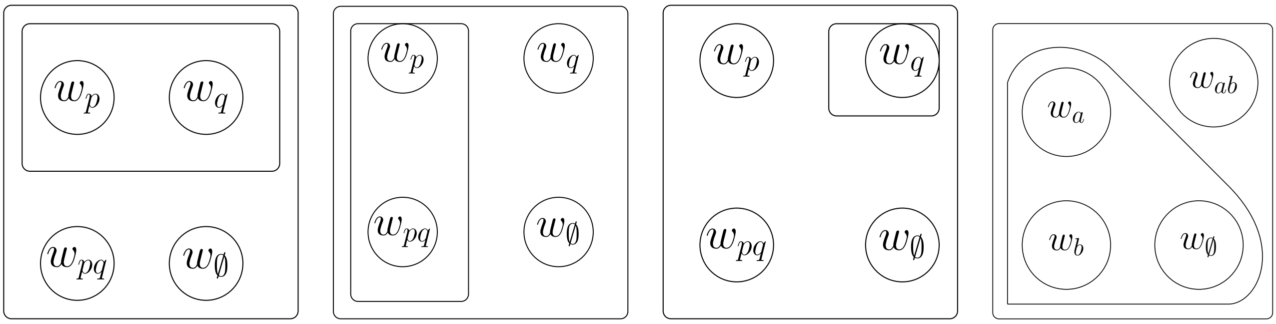
```
-- The pragmatic enrichment function [.]+ : ML -> BSML.
enrich :: MForm -> Form
enrich (MProp n)  = Prop n `And` NE
enrich (MNeg f)   = Neg (enrich f) `And` NE
enrich (MDia f)   = Dia (enrich f) `And` NE
enrich (MAnd f g) = (enrich f `And` enrich g) `And` NE
enrich (MOr f g)  = (enrich f `Or`  enrich g) `And` NE
```

To test the semantic effect of enrichment, and some other properties of ML as a fragment of BSML, we also implement the canonical embedding of ML into BSML.

```
-- Embedding ML >-> BSML
toBSML :: MForm -> Form
toBSML (MProp n)  = Prop n
toBSML (MNeg f)   = Neg (toBSML f)
toBSML (MAnd f g) = And (toBSML f) (toBSML g)
toBSML (MOr f g)  = Or (toBSML f) (toBSML g)
toBSML (MDia f)   = Dia (toBSML f)
```

As the reader should expect at this point, we also implement an arbitrary instance for formulas of ML, which is completely analogous to that for BSML:

```haskell
randomMProp :: Gen MForm
randomMProp = MProp <$> choose (1, numProps)

instance Arbitrary MForm where
  arbitrary = sized $ \case
    0 -> randomMProp
    _ -> oneof [
      randomMProp,
      MNeg <$> f,
      MAnd <$> f <*> f,
      MOr  <$> f <*> f,
      MDia <$> f]
    where f = scale (`div` 2) arbitrary

  shrink (MNeg f)     = f        :  [MNeg f'      | f'        <- shrink f]
  shrink (MDia f)     = f        :  [MDia f'      | f'        <- shrink f]
  shrink (MAnd f1 f2) = [f1, f2] ++ [MAnd f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink (MOr f1 f2)  = [f1, f2] ++ [MOr  f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink _            = []
```



(a) $\vDash (p \vee q)$, $\vDash [p \vee q]^+$ (b) $\vDash (p \vee q)$, $\vDash [p \vee q]^+$ (c) $\vDash (p \vee q)$, $\nvDash [p \vee q]^+$ (d) $\vDash (p \vee q)$, $\vDash [p \vee q]^+$

Figure 2

3.2 Free-Choice inference

The way FC-inference is now modelled, is that given a formula/statement like $\Diamond(\alpha \vee \beta)$ in ML, we enrich it to obtain a formula BSML:

$$[\Diamond(\alpha \vee \beta)]^+ = \Diamond((\alpha \wedge \texttt{NE}) \vee (\beta \wedge \texttt{NE})) \wedge \texttt{NE}$$

This formula more accurately represents the meaning of a sentence in linguistics, given speakers' distaste for null verification. To accurately model FC, we should then be able to infer $\Diamond\alpha \wedge \Diamond\beta$ from $[\Diamond(\alpha \vee \beta)]^+$.

3.3 Testing Enrichment and FC

We can use the library QuickCheck to test the enriched behavior of our implementation of BSML, to make sure we correctly implemented FC.

The following uses the HSpec library to define different tests. We use a mix of QuickCheck and specific inputs, depending on what we are testing for.

The "Figure 3" section in the tests corresponds to Figure 1 described above in a previous section of the paper. The "Motivating Example" uses the Figure 2 models in order to show our disjunction behaves as expected. The paper [?] describes some formulas that make use of enrichment that must hold in these models.

```haskell
main :: IO ()
main = hspec $ do
  describe "Figure 3" $ do
    it "Figure 3b, [<>(p ^ q)]+" $
      (m3b, s3b) |= enrich (MDia (ma `MAnd` mb)) `shouldBe` False
    it "Figure 3c, [<>(p v q)]+" $
      (m3c, s3c) |= enrich (MDia (ma `MOr` mb)) `shouldBe` True

  describe "Motivating Example" $ do
    it "(a) |= (a v b) == True" $
      (mM, sMA) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(a) |= [a v b]+  == True" $
      (mM, sMA) |= enrich (ma `MOr` mb) `shouldBe` True

    it "(b) |= (a v b) == True" $
      (mM, sMB) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(b) |= [a v b]+ == True" $
      (mM, sMB) |= enrich (ma `MOr` mb) `shouldBe` True

    it "(c) |= (a v b) == True" $
      (mM, sMC) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(c) |= [a v b]+ == False" $
      (mM, sMC) |= enrich (ma `MOr` mb) `shouldBe` False

    it "(d) |= (a v b) == False" $
      (mM, sMD) |= toBSML (ma `MOr` mb) `shouldBe` False
    it "(d) |= [a v b]+  == False" $
      (mM, sMD) |= enrich (ma `MOr` mb) `shouldBe` False

    prop "Arbitrary |= (a v b) !== Arbitrary |= [a v b]+  == False" $
      expectFailure $ \(TPM m s) -> (m,s) |= toBSML (ma `MOr` mb) == (m,s) |= enrich (ma `
          MOr` mb)
```

Narrow-scope and wide-scope relate to the "pragmatic enrichment function," and these tests confirm that FC-inference holds in our implementation. The flatness test confirms that our implementation of team semantics is flat on ML-formulas.

```haskell
  describe "Properties from Paper" $ modifyMaxSize (const 25) $ do
    prop "NarrowScope, <>(a v b) =| (<>a ^ <>b)" $
      \(TPM m s) -> (m,s) |= enrich (MDia (ma `MOr` mb)) == (m,s) |= enrich (MDia ma `MAnd`
           MDia mb)
    prop "Wide Scope, <>a v <>b) =| <>a ^ <>b" $
      \(TPM m s) -> all (\w -> rel' m w == s) s <= ((m,s)  |= enrich (MDia ma `MOr` MDia mb
          ) <= (m,s) |= enrich (MDia ma `MAnd` MDia mb))

  describe "Flatness" $ modifyMaxSize (const 15) $ do
    modifyMaxSize (const 10) $ prop "(M,s) |= f <==> M,{w} |= f forall w in s" $
      \(TPM m s) f -> (m,s) |= toBSML (f::MForm) == all (\w -> (m, [w]) |= toBSML f) s
    prop "M,{w} |= f <==> M,w |= f" $
      \(WPM m w) f -> (m, [w]) |= toBSML (f::MForm) == (m,w) |= f
    prop "Full BSML is *not* flat" $ expectFailure $
      \(TPM m s) f -> (m,s) |= (f::Form) == all (\w -> (m, [w]) |= f) s
```

We also test our ability to correctly parse and pretty print, however the specifics of these functions are explained in the following sections.

```haskell
  describe "Pretty Print and Parsing" $ do
    prop "formula -> prettyPrint -> Parsing == original formula" $
      \f -> parseFormula (ppForm (f::Form)) == Right f
  where
    ma = MProp 1
    mb = MProp 2
```

To run the all the tests, use `stack test`.

4 Lexing and Parsing BSML Formulae

4.1 Lexing

Below we describe our code for lexing and parsing BSML formulas. We use the well-known tools Alex and Happy for lexing and parsing, respectively. This code is partially inspired by [?].

We start by telling Alex how to recognize certain (series of) symbols in the string we will be parsing and turn them into the corresponding token. These tokens are symbols that are assigned some meaning while a string is being parsed. Our set of tokens are defined in a separate file, `Token.hs`, which is omitted here since it is utterly uninspiring.

```
{
module Lexer where
import Token
}


%wrapper "posn"
$dig = 0 -9 -- digits
tokens:-
-- ignore whitespace and comments :
$white + ;
" --" .* ;
-- keywords and punctuation :
"("   { \ p _ -> TokenOB p }
")"   { \ p _ -> TokenCB p }
"_|_" { \ p _ -> TokenBot p }
"NE"  { \ p _ -> TokenNE p}
"[]"  { \ p _ -> TokenBox p }
"<>"  { \ p _ -> TokenDmd p }
"~"   { \ p _ -> TokenNot p }
"&"   { \ p _ -> TokenCon p }
"v"   { \ p _ -> TokenDis p }
-- Integers and Strings :
$dig +  { \ p s -> TokenInt ( read s ) p }
[P - Z] { \ p s -> TokenPrp ( ord ( head s ) - ord 'P') p }
[p - z] { \ p s -> TokenPrp ( ord ( head s ) - ord 'p' + 11) p }
```

This file, `Lexer.x` is *not* a valid Haskell file, it is only meant to be input to Alex. A user may run `alex Lexer.x` to generate `Lexer.hs`, a fully functioning lexer for the above tokens built by Alex.

4.2 Parsing

We now take a look at the code for the parser, which works using Happy. We first import the modules, including `Lexer`, that we require to run the parser.

```
> {
> module Parser where
>
> import Data.Char
> import Token
> import Lexer
> import Syntax
> import Semantics
> import ML
> }
```

Below, we describe the tokens that Happy needs to keep track of while reading the string. These tokens are congruent with those in the module Lexer.

```
> %name formula
> %tokentype { Token AlexPosn }
> %monad { ParseResult }
>
> %token
> BOT { TokenBot _ }
> NE {TokenNE _}
> AND { TokenCon _ }
> OR { TokenDis _ }
> NOT { TokenNot _ }
> BOX { TokenBox _ }
> DMD { TokenDmd _ }
> GDIS {TokenGDis _}
> NUM { TokenInt $$ _ }
> '(' { TokenOB _ }
> ')' { TokenCB _}
```

We describe also the binding hierarchy for binary operators in our language. The order of precedence is described by listing operators from weakest to strongest, as evidenced below. Note that the binding for all unary operators is stronger than the binding for binary operators, and unary operators operate at the same binding strength. This behaviour keeps in line with the way we would like to parse formulas in our language.

```
> %left OR
> %left AND
> %left GDIS
>
> %%
```

We now detail two different types of formulas - bracketed formulas and non-bracketed formulas. The reasoning for the distinction is rather simple - the presence of parentheses around a formula requires that any operations within the parentheses need to be given priority over operations outside of them. This may break regular precedence rules, and hence need to be accounted for.

```
> Form :: { Form }
> Form : BrForm { $1 }
> | Form AND Form { And $1 $3 }
> | Form OR Form { Or $1 $3 }
> | Form GDIS Form {Gor $1 $3}
> BrForm :: { Form }
> BrForm : NUM { Prop $1 }
> | BOT { Bot }
> | NE { NE }
> | NOT BrForm { Neg $2 }
> | BOX BrForm { Syntax.box $2 }
> | DMD BrForm { Dia $2 }
> | '(' Form AND Form ')' { And $2 $4 }
> | '(' Form OR Form ')' { Or $2 $4 }
> | '(' Form GDIS Form ')' { Gor $2 $4}
```

Next, we define error messages for our parser, as in [?]. These error messages describe where the error occurs exactly in the string, and why Happy failed to parse it.

```
> {
> data ParseError = ParseError { pe_str :: String
>                              ,pe_msg :: String
>                              ,pe_col :: Int}
>      deriving (Eq , Show)
>
> type ParseResult a = Either ParseError a
>
>
> happyError :: [ Token AlexPosn ] -> ParseResult a
> happyError [] = Left $
>      ParseError { pe_str = " " , pe_msg = " Unexpected end of input : " , pe_col = -1}
```

```
> happyError ( t : ts ) = Left $
>      ParseError { pe_str = " " , pe_msg = " Parse error : " , pe_col = col }
>      where ( AlexPn abs lin col ) = apn t
>
> myAlexScan :: String -> ParseResult [ Token AlexPosn ]
> myAlexScan str = go ( alexStartPos , '\n' ,[] , str )
>   where
>       go :: AlexInput -> ParseResult [ Token AlexPosn ]
>       go inp@( pos ,_ ,_ , str ) =
>           case alexScan inp 0 of
>               AlexEOF -> Right []
>               AlexError (( AlexPn _ _ column ) ,_ ,_ , _ ) -> Left $
>                   ParseError { pe_str = str , pe_msg = " Lexical error : " , pe_col =
  column -1}
>               AlexSkip inp' len -> go inp'
>               AlexToken inp' len act -> go inp'  >>=
>                   (\ x -> Right $ act pos ( take len str ) : x )
```

Finally, we describe the actual parsing function itself, called `parseFormula`. Upon running `happy Parser.ly`, we get a Haskell file `Parser.hs` which contains the parseFormula function. The output for parseFormula is of type `Either ParseError Form`, since parser might fail (on invalid input).

```
> parseFormula :: String -> ParseResult Form
> parseFormula str = go $ myAlexScan str  >>= formula
>   where
>       go ( Left err ) = Left $ err { pe_str = str }
>       go ( Right res ) = Right res
>
> }
```

5 An Executable Function

We now describe a simple executable function. Upon running `main`, the function asks the user to provide a BSML formula as input. Using the parser, we parse this string, and try to come up with an example that does not satisfy the formula.

Note that as with any other QuickCheck based testing suite, the function's inability to find a counter-example does not suggest that one does not exist! Below, we have a function that runs 100 tests (as standard, as we will soon see), which is usually sufficient to find a counter-example.

```
module Main where
import Syntax
import Semantics
import Parser

import Test.QuickCheck
import Text.Read (readMaybe)
```

After importing the necessary modules, we describe two small helper functions. The first is simply for notational convenience: we use it to check whether a given model falsifies a given formula.

The second uses QuickCheck to generate arbitrary models, and terminates when it finds a model falsifying the given formula.

```
falsifies :: TeamPointedModel -> Form -> Bool
falsifies (TPM m s) f = not ((m, s) |= f)

findCounterexample :: Form -> Int -> IO ()
findCounterexample f n = quickCheck (withMaxSuccess n (`falsifies` f))
```

The main function asks the user to provide a BSML formula as a string. It first checks that the string represents a well-formed formula using the parser, and then uses the helper functions above to produce (if it can find such an example, of course!) a model and a team that falsify the formula provided by the user. The user is also asked to specify the maximum number of tests they wish to run within the function. The default number of tests is fixed at 100.

```
main :: IO ()
main = do
    putStrLn "Enter a BSML formula whose validity you want to check:"
    input <- getLine
    case parseFormula input of
        Left s -> putStrLn(pe_str s ++ pe_msg s ++ show(pe_col s))
        Right f -> do
            putStrLn "Type in the maximum number of tests you wish to run (a non-number
                will result in 100 tests being run):"
            testnum <- getLine
            case (readMaybe testnum :: Maybe Int) of
                Nothing -> findCounterexample f 100
                Just x -> findCounterexample f x
```

6 Natural Deduction

This section explains the implementation of Natural Deduction proofs for BSML. Our code should still be considered *Work in progress*, but already has useful functionality, with clear potential for future extensions and/or improvements.

Since we want users of this module to solely be able to construct proofs using the supplied axioms, we explicitly name the exports of this module and omit the constructor for the `Proof`-type.

```
module ND
  (
    Proof
  , sorry
  , assume

  -- Rules
  .
  .
  .
  ) where
```

To represent an ND-proof, we use the `Proof`-type, which stores the conclusion of the proof and all of its open (non-discharged) assumptions. We use a `Set` to represent proofs to allow easy omission of duplicates and removal (discharges) of assumptions. For the sake of simplicity, we decided not to store the entire ND-tree leading to a conclusion, but it would be a straightforward adaptation to make, if desired.

```
import Syntax

import Data.List
import Data.Set (Set)
import qualified Data.Set as Set

-- Type for representing ND-proofs, constructor Prf is for internal use only!
data Proof = Prf {conclusion :: Form,
                  assumptions :: Set Form}
  deriving (Show)

-- Pretty printing for proofs
ppProof :: Proof -> String
ppProof (Prf f fs) = intercalate ", " ss ++ " |- " ++ ppForm f
  where ss = ppForm <$> Set.toList fs
```

Next, we define a function to represent making a new assumption in a proof; given any formula φ, it returns the proof with conclusion φ and open assumptions $\{\varphi\}$.

```
assume :: Form -> Proof
assume = Prf <*> Set.singleton
```

Further, we define some functions for convenience. Here, `sorry` completely subverts our system by creating a proof for any conclusion and set of assumptions, but it can be useful for users as a placeholder value in proofs. It is similar to `Lean`'s sorry, and triggers a warning anytime it is used. The `hasNE` and `hasGor` functions check whether a formula uses the `NE` or `Gor` constructor anywhere, which is needed for checking some side-conditions on ND-rules. Recall that `hasCr` was defined in ??.

```
sorry :: Form -> Set Form -> Proof
sorry = Prf
{-# WARNING sorry "Proof uses sorry!" #-}

-- Used for checking side-conditions

hasNE :: Form -> Bool
hasNE = hasCr _NE

hasGor :: Form -> Bool
hasGor = hasCr _Gor
```

Now, we can define all of the rules axiomatizing BSML, as proven in [?] (see Chapter 4). There are quite a lot of rules (32 to be exact, we implemented all of them!), so we will not show all of them here, but we will highlight a few to give a good idea of the implementations.

Take e.g. the rule ∨I, which introduces $\varphi \vee \psi$ from a proof of φ under the condition that ψ does not contain `NE`. This is implemented as:

```
-- (c) Rules for v

orIntroR :: Form -> Proof -> Proof
orIntroR g (Prf f ass)
  | hasNE g    = error "Cannot vIntro a formula containing NE!"
  | otherwise = Prf (Or f g) ass
```

For a more complicated example, we can consider ∨E, the rule for disjunction-elimination:

```
orElim :: Proof -> Proof -> Proof -> Proof
orElim (Prf (Or f g) ass) (Prf h ass1) (Prf h' ass2)
  | h /= h'          = error "Cannot apply vElim, conclusions of latter proofs do not match!"
  | any hasNE ass' = error "Cannot apply vElim, latter proofs have undischarged assumptions
      containing NE!"
  | hasGor h         = error "Cannot apply vElim, latter conclusion contains V/."
  | otherwise        = Prf h $ ass <> ass'
  where ass' = Set.delete f ass1 <> Set.delete g ass2
orElim _ _ _ = error "Cannot apply vElim, conclusion of first proof is not a disjunction!"
```

Note in particular how we use a combination of guards and pattern matching to ascertain whether the formulas have the correct form and that all the side-conditions are met.

For the sake of completeness, we wil also show □Mon, a rule involving modal operators and an interesting side condition: for every assumptions φ of the first proof, $\Box\varphi$ should be the conclusion of one of the latter proofs. Also note how we use `foldMap` to extract the assumptions from each of the latter proofs and take their union.

```
boxMon :: Proof -> [Proof] -> Proof
boxMon (Prf g ass) ps
  | Set.map box ass `Set.isSubsetOf` Set.fromList (map conclusion ps) =
    Prf (box g) $ foldMap assumptions ps
  | otherwise = error "Cannot apply []Mon, former proof has undischarged assumptions!"
```

6.1 Some examples

As mentioned before, the implementation of natural deduction is still a *Work in progress*, but in this section, we will quickly illustrate some of the functionality we already have.

```
module ND_examples where

import Syntax
import ND
```

The most obvious functionality is obvious: representing proofs! For example, here is the proof for ex falso in our language:

```
-- Given some f, gives a proof of _||_ |- f, as in Lemma 4.2 of Aloni2024.
exFalso :: Form -> Proof
exFalso f = botbotCtr f $ orIntroR Bot $ assume botbot
```

One of the drawbacks of our language is that the type signature of proofs is not very illuminating as to their content. However, note that the information can be retrieved by inspecting the proof:

```
ghci> ppProof $ exFalso $ Prop 1
"(_|_ & NE) |- 1"
```

For a more involved proof, consider the following implementation of the ND-proof for distributivity of $\wedge$ over $\vee$:

```
-- Distributivity of And over Or: f & (g v h)      (f & g) v (f & h)
-- under the condition that f is NE-free
andOrDistr :: Form -> Form -> Form -> Proof
andOrDistr f g h =
  orElim
    (andElimR ass)
    (orIntroR (f `And` h) $ andIntro pf $ assume g)
    (orIntroL (f `And` g) $ andIntro pf $ assume h)
  where
    ass = assume $ f `And` (g `Or` h)
    pf = andElimL ass
```

While this gives a structured way of writing proofs that are guaranteed to be sound, it is still missing quite some functionality one would expect from an interactive theorem prover: for example, automatically filling in holes in a proof using `sorry`.

Apart from representing proofs, we can also create new functions between proofs from our axioms. These represent admissible rules in the system, for example:

```
-- orIntroL is admissible
orIntroL :: Form -> Proof -> Proof
orIntroL = (orComm .) . orIntroR
```

shows that $\vee$I on the left (instead of the right, as in the axiomatization), is admissible.

7 Future work

In the future, we would like to add an easier-to-navigate UI, which would help people interested in BSML but not well-versed in Haskell use our model-checker. A symbolic model-checker would also be nice to have, especially if we could additionally display models to the user with `graphviz`. Something similar to [?] for BSML would be a wonderful goal.

Among the more fascinating (and ambitious) directions for future work would be extending the Natural Deduction representation we have described to automate proof-search. Especially given that BSML is such a new logic, there are bound to be some fantastic results within it, and having automated proof-search would be a very powerful tool with which to explore these horizons.

Until then, we hope you have enjoyed using our model-checker as much as we loved making it! Special thanks to Maria for her wonderful paper, and to Malvin for providing an environment where we could explore and learn something rather wonderful and exciting - it was a joy to work on the project, and we hope that it is a meaningful contribution to a fascinating field of study.

References

[AAY24] Maria Aloni, Aleksi Anttila, and Fan Yang. State-based modal logics for free choice. *Notre Dame Journal of Formal Logic*, 65(4), November 2024.

[Alo22] Maria Aloni. Logic and conversation: The case of free choice. *Semantics and Pragmatics*, 15(5):1–60, June 2022.

[BRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[Dvo18] Kajetan Dvoracek. Natural Deduction Prover, June 2018.

[GVLM24] Malvin Gattinger, Stefano Volpe, Bas Laarakker, and Daniel Miedema. Smcdel — a symbolic model checker for dynamic epistemic logic, 2024. Version 1.3.0.