Intuitionistic Propositional Logic Tableaux in Haskell

Valentino Filipetto & Gian Marco Osso

Friday 4th February, 2022

Abstract

We present an implementation in Haskell of a tableau-style proof system for the implication-free fragment of intuitionistic propositional logic, drawing inspiration from a preexisting implementation of classical first-order logic proposed here [3]. We also propose a tableau prover for classical propositional logic, obtained via a minor modification of the original one. These two can be used in conjunction to have experimental evidence of Glivenko's theorem.

Contents

1 Theory

We used [1] as a reference for the tableau rules. In the following we shall give an overview of the calculus.

The calculus exploits signed formulas, in particular it uses three signs: T, F, F_C. $T\varphi$ intuitively stands for "we have a proof of φ", $F\varphi$ stands for "we don't have a proof of φ at the moment" and $F_C\varphi$ stands for "φ has been refuted". To better understand what this means we can consider the semantics for intuitionistic logic based on intuitionistic Kripke models (IKVs in short). An intuitionistic Kripke model is a triple (W, R, V) such that:

- W is a nonempty set,
- R is a weak partial order, i.e. a transitive, reflexive and antisymmetric relation,
- V is a function $V : Prop \to \mathcal{P}(W)$ such that for all $w, w' \in W$ and all $p \in Prop$ $w \in V(p) \wedge wRw'$ implies $w' \in V(p)$ (i.e. the range of V is given by the set of upsets of (W, R)).

One can then define a translation from intuitionistic propositional formulas into basic modal logic formulas as follows, given an intuitionistic propositional formula φ,

- if $\varphi = p$, $T(\varphi) = p$
- if $\varphi = \chi \wedge \psi$, $T(\varphi) = T(\chi) \wedge T(\psi)$
- if $\varphi = \chi \vee \psi$, $T(\varphi) = T(\chi) \vee T(\psi)$
- if $\varphi = \psi \to \chi$, $T(\varphi) = \Box(T(\psi) \to T(\chi))$
- if $\varphi = \neg\psi$, $T(\varphi) = \Box(\neg T(\psi))$

One finally obtains a semantics for intuitionistic logic by stipulating that, given an intuitionistic Kripke model (W, R, V) and $w \in W$, an intuitionistic formula φ holds at w ($w \vDash \varphi$) if and only if $w \vDash T(\varphi)$ in the standard Kripke semantics. Accordingly, an intuitionistic formula φ is valid if it holds at all worlds in every intuitionistic Kripke model, and it is satisfiable if there's some IKV containing a world where φ holds.
Given an intuitionistic propositional formula φ, an IKV (W, R, V) and $w \in W$, one can think of the sign of φ at w as follows:

- φ has sign T if $w \vDash \varphi$,
- φ has sign F if it is not the case that $w \vDash \varphi$,
- φ has sign F_C if $w \vDash \neg\varphi$.

T rules	F rules	$F_C$rules
$\frac{S,T(\varphi\wedge\psi)}{S,T\varphi,T\psi}$	$\frac{S,F(\varphi\wedge\psi)}{S,F\varphi/S,F\psi}$	$\frac{S,F_C(\varphi\wedge\psi)}{S_c,F_C\varphi/S_c,F_C\psi}$
$\frac{S,T(\varphi\vee\psi)}{S,T\varphi/S,T\psi}$	$\frac{S,F(\varphi\vee\psi)}{S,F\varphi,F\psi}$	$\frac{S,F_C(\varphi\vee\psi)}{S,F_C\varphi,F_C\psi}$
$\frac{S,T(\neg\varphi)}{S,F_C\varphi}$	$\frac{S,F(\neg\varphi)}{S_c,T\varphi}$	$\frac{S,F_C(\neg\varphi)}{S_c,T\varphi}$

The rules which are proposed in [1] are the following:

where we have left out rules for implication as we restricted our implementation to the implication-free fragment of intuitionistic propositional logic. This choice was made as the rules presented in [1] for intuitionistic implication make it possible to have infinite loops, whereas other sources such as [2] present more than one rule for implication, depending on the structure of the antecedent and consequent. In both cases, it would have been excessively complicated (and probably not very interesting from the logical and programming point of view) to obtain a complete and always terminating system.

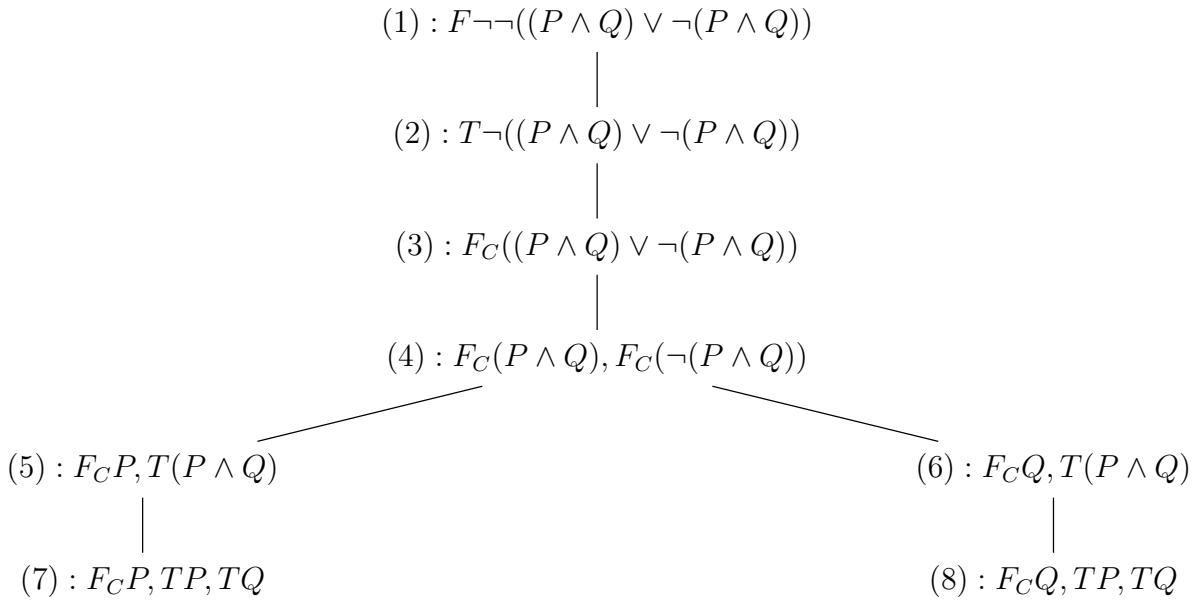
We divide these nine rules into different groups:

1. Rule$_1$: $T\wedge$,$F\vee$ and $F_C\vee$; they don't lead to splitting and the formula is decomposed into its components, which keep their original sign.
2. Rule$_2$: $T\vee$ and $F\wedge$; they lead to splitting and each of the resulting branches keeps a subformula, with the original sign.
3. Falseneg: $F\neg$ and $F_C\neg$; they bring us from S to S_c, and the resulting subformula is a T formula.
4. Trueneg: $T\neg$; it deletes negation and it changes the sign to F_C.
5. Prfalseconj: $F_C\wedge$ (provably false conjunction); it leads to splitting and cancellation of previous false formulas. The subformulas keep the same sign.

Rules in Rule$_1$ and Rule$_2$ are intuitive. The rules for false/provably false negation and for provably false conjunction both lead from a set of formulae S to another set, S_c, which is $S \setminus \{F\varphi | F\varphi \in S\}$. The idea is that every time we apply such rules we have to delete from our set of remaining formulas all false formulas. This leads to a problem: if one has more than one pending F formula, and one decides to expand a tableau node by applying a Falseneg or Prfalseconj rule, one could delete formulas which were needed for the tableau branch to close. Therefore it is necessary to explore all possibilities, i.e. applying the rules to the formulas in all the orders possible. This problem will be reflected on the implementation, which becomes slightly harder than one would expect.

There's a close correspondence between these rules and the Kripke model semantics for intuitionistic logic, which is useful to understand how the prover works. As it usually is for tableaux proofs, one wants to establish the fact that a formula is valid by trying to build a countermodel. In our particular case, the expansion of a branch is mirrored in a step by step construction of a Kripke model. Given a formula φ, if one branch in its tableau is open (i.e. it does not contain contradicting signed literals) then one use it to build a rooted IVK in which the root refutes φ. Conversely, if all branches of the tableau are closed, then one cannot find any world in any IVK where φ does not hold.

Consider the following example of a tableau proof of the intuitionistic tautology $\varphi = \neg\neg((P \wedge Q) \vee (\neg(P \wedge Q))$



One can read the tableau above as an attempt at building a Kripke model where the formula φ does not hold at the root node (1). The rules then become much more intuitive as (keeping in mind the translation to modal logic given above, which can be summarized as "$\neg$ is $\Box\neg$")

- if φ does not hold at node (1), then node (1) must have at least a successor (2) where $\varphi_1 = \neg((P \wedge Q) \vee \neg(P \wedge Q))$ holds,
- if φ_1 holds at node (2) then all of its successors must consider $(P \wedge Q) \vee \neg(P \wedge Q)$ provably false, in particular there must be some successor node (3) (which may actually just be node (2)) which can be marked with $F_C((P \wedge Q) \vee \neg(P \wedge Q))$,
- similarly to the point above, there must be a successor node (4) (again, node (2) itself works just as well) which can be marked with both $F_C(P \wedge Q)$ and $F_C(\neg(P \wedge Q))$,
- since $F_C(P \wedge Q)$ holds in node (4), it must be that either (4) has one successor i such that $i \vDash \neg P$ or it has one successor j such that $j \vDash \neg Q$. Consequently one builds successor nodes (5) and (6) and labels them appropriately,

- since $P \wedge Q$ holds at node (5), it must be that both P and Q hold at all successors of (5), so there must be a node (7) (which again could be (5) itself) which can be marked as $F_C P, TP, TQ$, so P is both true and provably false in it. This is a contradiction, which means that (4) cannot have successor nodes which can be marked with $F_C P$.
- In a way entirely similar to the point above, considerations on node (6) show that (4) cannot have a successor where $F_C Q$ holds.
- This means that no node in any IVK can ever be marked in the same way as (4), so backtracking shows that the same holds for nodes (3), (2) and (1). In particular one obtains that $\neg\neg((P \wedge Q) \vee \neg(P \wedge Q))$ is an intuitionistic validity.

In order to appreciate some other features of the proof system, one can consider the following tableau for the non-valid formula $\psi = P \vee \neg P$.

$$
\begin{array}{c}
(1) : F(P \vee \neg P) \\
\Big| \\
(2) : FP, F(\neg P) \\
\Big| \\
(3) : TP
\end{array}
$$

The branch does not close, and this is in line with the fact that ψ is not intuitionistically valid. What might seem strange at first sight is that, in passing from node (2) to node (3), one has to delete the signed formula FP. This is best understood when one switches to the Kripke model perspective. Again, consider a model where ψ does not hold at node (1). Then it must be that (1) has a successor (2) (which as above, can very well be (1) itself) where both P does not hold and $\neg P$ does not hold. The fact that $\neg P$ does not hold at (2) just means that there is *some* successor (3) where P holds. On the other hand, the fact that P does not hold at (2) does not give any guarantees on whether it holds at successors of (2) or not. Therefore a model based on a frame given by the tableau, with valuation V such that $V(p) = \{3\}$ is a model where ψ holds at the root node. This shows that ψ is not an intuitionistic validity. Note that, in case more than one F formula is present at a tableau node, it may be necessary to duplicate tableaux and check whether at least one of the possible expansions leads to a closed tableau.

2 Implementation

2.1 Formulas

```
module Formulas where

import.Data.List
{-# LANGUAGE BlockArguments #-}
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
```

We represent indices and formulas as

```
type Index = [Int]

data Frm =  P Int
         | N Frm
         | C Frm Frm
         | D Frm Frm
     deriving (Eq,Ord)

data Sign = T | F | Fc deriving Eq

newtype SFrm = S (Sign, Frm)

instance Eq SFrm where
  S (x,y) == S (z,v) = (x,y)==(z,v)

subf :: Frm -> [Frm]
subf (P x) = [P x]
subf (N f) = [f]
subf (C f g) = [f,g]
subf (D f g) = [f,g]
```

The subf function is a tool that will be needed for tableau expansion.

2.2 Tableaux

```
module Tableau where

import Formulas
```

We represent a node in the following way. Each node is composed of six lists, the list of true atoms (Tp), of false atoms (Fp), of refuted atoms (F_Cp), the list of pending formulas that don't lead to any deletion of F formulas in the set S; the last two lists are a list of pending formulas with the shape $F(\neg\varphi)$ and a list that only contains pending formulas with the shape $F_C(\neg\varphi)$ and $F_C(\varphi \wedge \psi)$. A tableau is then just a list of nodes: in particular, they are stored as the list corresponding to their "leaf level", as that encodes all the necessary information.

```
data Node  = Nd Index [Frm] [Frm] [Frm] [SFrm] [SFrm] [Sfrm]
instance Eq Node where
  Nd i p n f tp fp fcp == Nd j d s a ta fa fca = i==j && p==d && n==s && f==a && tp==ta &&
      fp==fa && fcp ==fca
type Tableau = [Node]
```

The reason for having three separate lists of pending formulas is that we want to distinguish them based on how tricky their associated rule is. In particular, we know that applying the rules to formulas of the form $F\neg$, $F_C\neg$ and $F_C\wedge$ leads to cancellation of all remaining F formulas and need to be treated carefully in order to preserve completeness of the system.

2.3 Helper functions for step

The following are helper functions which are needed in the definition of step, they are pretty much self explainatory

```
module HelperFunctions

import Formulas
import Tableau

rule1 :: SFrm -> Bool
rule1 (S (T, C _ _))  = True
rule1 (S (F, D _ _))  = True
rule1 (S (Fc, D _ _)) = True
rule1 _               = False

rule2 :: SFrm -> Bool
rule2 (S (T, D _ _))  = True
rule2 (S (F, C _ _))  = True
rule2 _               = False

falseneg :: SFrm -> Bool
falseneg (S (F, N _))  = True
falseneg (S (Fc, N _)) = True
falseneg _             = False

prfalseconj :: SFrm -> Bool
prfalseconj (S (Fc, C _ _)) = True
prfalseconj _               = False

trueneg :: SFrm -> Bool
trueneg (S (T, N _)) = True
trueneg _            = False


deletewarning :: SFrm -> Bool
deletewarning (S (x, y)) = prfalseconj (S (x, y)) || falseneg (S (x, y))

tlit, flit, fclit :: SFrm -> Bool
tlit (S (T, P _))    = True
tlit _               = False
flit (S (F, P _))    = True
flit _               = False
fclit (S (Fc, P _))  = True
fclit _              = False

makesign :: Sign -> Frm -> SFrm
makesign x y = S (x,y)
makenegative :: Frm -> SFrm
makenegative f = S (F,f)
maketrue :: Frm -> SFrm
maketrue f = S (T,f)
makefalse :: Frm -> SFrm
makefalse f= S (Fc, f)

removesign :: SFrm -> Frm
removesign (S (_, f)) = f

signof:: SFrm -> Sign
signof (S (T, _ )) = T
signof (S (F, _ )) = F
signof (S (Fc, _)) = Fc
```

2.4 The step rule

The step rule is what allows us to develop our tableau in that it tells us what to do based on what we have inside the lists of a specific node. However, it is very long and cumbersome, so we only explain the most important parts of it.

As we have seen, a node is composed of six lists, and the step rule is essentially divided into three parts accordingly: the first acts on the list of pending formulas that don't lead to any deletion of F formulas (i), the second one on the list of pending formulas with the shape $F(\neg\varphi)$ (ii) and finally the third one on the last list, i.e. the list that only contains pending formulas with the shape $F_C(\neg\varphi)$ and $F_C(\varphi \wedge \psi)$ (iii). The rule then behaves in the following way: it first tries to develop all formulas in (i). Then, when (i) is empty, it switches to (ii), and when (ii) is empty as well it switches to (iii). The reason for this is that we want to avoid trying all possible combinations of rule application (which might be necessary since we have rules that, when applied, delete all F formulae), therefore we want to develop every unproblematic F formula before we turn to formulas whose treatment might delete other F formulas.

The following is part of the code that is used to treat the list (i). Let's take the case in which f is a true atom, as an example. What the rule does is: we add f to the list of true atoms only if it is not part of F atoms or F_C atoms. Obviously, if this is not the case, we have a contradiction, and we get a tableau containing just the empty list. This will come in handy later.

```
module Step where

import Formulas
import Tableau
import HelperFunctions

step :: Node  -> [Tableau]
step (Nd i positives negatives falses [] [] []) = [[Nd i positives negatives falses [] []
    []]]
step (Nd i positives negatives falses (f:fs) fnegpending fcpending)
  | tlit  f = [[Nd i (removesign f:positives) negatives falses fs fnegpending fcpending |
      not (elem (removesign f) negatives || elem (removesign f) falses)]]
  | flit  f = [[Nd i positives (removesign f :negatives) falses fs fnegpending fcpending |
      removesign f `notElem` positives]]
  | fclit  f = [[Nd i positives negatives (removesign f :falses) fs fnegpending fcpending |
       removesign f `notElem` positives]]
```

Let's take a more complicated case, i.e. rule1. We make a case distinction: we can either have a true conjunction, a false disjunction or a provably false disjunction. In the last two cases there is a technicality, i.e. we have to make sure that we put the components in the right list. Take the last case, as an example. The subformulas can either lead to no deletion or – because they are either of the shape $F_C(\varphi \wedge \psi)$ or of the shape $F_C(\neg\varphi)$ – to a deletion. Hence we have to put them in the proper list accordingly.

```
  | rule1 f = if signof f == T then [[Nd i positives negatives falses ([maketrue y | y <-
      subf $ removesign f]++fs) fnegpending fcpending]]
               else if signof f == F then [[Nd i positives negatives falses ([makenegative y
                    | y <-subf $ removesign f, not (deletewarning ( makenegative y))]++fs)
                   ([makenegative y | y <-subf $ removesign f, deletewarning (makenegative y
                    ) ]++fnegpending) fcpending]]
               else [[Nd i positives negatives falses ([makefalse y | y <-subf $ removesign
                   f, not (deletewarning (makefalse y)) ]++fs) fnegpending ([makefalse y | y
                    <-subf $ removesign f, deletewarning  (makefalse y) ]++fcpending)]]
  | rule2 f = if signof f == T then [[Nd (i++[0]) positives negatives falses (maketrue (
```

```
      head (subf $ removesign f)): fs) fnegpending fcpending, Nd (i++[1]) positives
      negatives falses (map maketrue (tail (subf (removesign f))) ++ fs) fnegpending
      fcpending ]]
              else [[Nd (i++[0]) positives negatives falses ([makenegative (head (subf (
                  removesign f))) | not (deletewarning (makenegative (head (subf (
                  removesign f))))) ]++fs) ([makenegative (head (subf (removesign f))) |
                  deletewarning (makenegative (head (subf (removesign f)))) ]++fnegpending)
                   fcpending, Nd (i++[1]) positives negatives falses ([makenegative (last (
                  subf (removesign f))) | not (deletewarning (makenegative (last (subf (
                  removesign f))))) ]++fs) ([makenegative (last (subf (removesign f))) |
                  deletewarning (makenegative (last (subf (removesign f)))) ]++fnegpending)
                   fcpending]]
  | falseneg f = [[Nd i positives [] falses [maketrue (head (subf (removesign f)))] []
      fcpending ]]
  | prfalseconj f = [[Nd (i++[0]) positives [] falses [makefalse (head (subf (removesign f)
      )) | not (deletewarning (makefalse (head (subf (removesign f)))) )] [] ([makefalse (
      head (subf (removesign f))) | deletewarning (makefalse (head (subf (removesign f))))
      ]++ fs ), Nd (i++[1]) positives [] falses [makefalse (last (subf (removesign f))) |
      not (deletewarning (makefalse (last (subf (removesign f)))) )] [] ([makefalse (last (
      subf (removesign f))) | deletewarning (makefalse (last (subf (removesign f)))) ]++fs)
       ]]
  | trueneg f = [[Nd i positives negatives falses ([makefalse (head (subf (removesign f)))
      | not (deletewarning (makefalse (head (subf (removesign f)))) )]++fs) fnegpending ([
      makefalse (head (subf (removesign f))) | deletewarning (makefalse (head (subf (
      removesign f)))) ]++fcpending) ]]
```

What the step rule does on (ii) is slightly different, because – given that (ii) is a list of formulas $\varphi_1, \ldots, \varphi_n$, where φ_i has the shape $F(\neg\varphi)$ – it outputs two tableaux: the first one is the result of applying the $F\neg$ rule to φ_1, while the second one is the list containing the original node in which φ_1 has been deleted from (ii), i.e. (ii) consists of $\varphi_2, \ldots, \varphi_n$. Of course this behaviour is repeated similarly for this last list, so we again generate two tableaux, one for φ_2 and another one for $\varphi_3, \ldots, \varphi_n$, and so on. The reason is that, whenever we develop one of these φ_i's, we automatically lose all the others in the list, as well as all F literals (after all, they are all F formulae), hence – for a list of n formulas – there are exactly n cases to explore, which are the n cases where one deletes φ_i first.

```
step (Nd i positives _ falses [] (f:fs) fcpending)
  | tlit  f = undefined
  | flit  f = undefined
  | fclit  f = undefined
  | rule1 f = undefined
  | rule2 f = undefined
  | falseneg f = [[Nd i positives [] falses [maketrue (head (subf (removesign f)))] []
      fcpending ] , [Nd i positives [] falses [] fs fcpending]]
  | prfalseconj f = undefined
  | trueneg f = undefined
```

Lastly, the step rule works on list (iii), i.e. a list that only contains either formulae of shape $F_C(\varphi \wedge \psi)$ or of shape $F_C(\neg\varphi)$. Because when we arrive at this stage we have no formulae in (ii) and we can only delete F literals, and moreover the rules for $F_C(\varphi \wedge \psi)$ and $F_C(\neg\varphi)$ don't lead to any deletion of formulae inside (iii), in this case we don't need the machinery we saw in (ii), we can just tackle them sequentially as we have seen for (i). Besides, this is also the reason why we kept this kind of formulae as the last "kind" to be treated by the algorithm.

So, this is the last part of the step rule:

```
step (Nd i positives _ falses [] [] (f:fs))
  | tlit  f = undefined
  | flit  f = undefined
  | fclit  f = undefined
```

```
  | rule1 f = undefined
  | rule2 f = undefined
  | falseneg f = [[Nd i positives [] falses [maketrue (head (subf (removesign f)))] [] fs
      ]]
  | prfalseconj f = [[Nd (i++[0]) positives [] falses [makefalse (head (subf (removesign f)
      )) | not (deletewarning (makefalse (head (subf (removesign f)))) )] [] ([makefalse (
      head (subf (removesign f))) | deletewarning (makefalse (head (subf (removesign f))))
      ]++ fs ), Nd (i++[1]) positives [] falses [makefalse (last (subf (removesign f))) |
      not (deletewarning (makefalse (last (subf (removesign f)))) )] [] ([makefalse (last (
      subf (removesign f))) | deletewarning (makefalse (last (subf (removesign f)))) ]++fs)
       ]]
  | trueneg f = undefined
```

2.5 Solving Tableaux

The solve function is the function that we use to determine theoremhood of a formula. Since the rules for the implication-free fragments of intuitionistic logic and classical logic coincide, it can be used for determining theoremhood of formulas in any of the two systems.

```
module Solve where
import Formulas
import Tableau
import Step
import HelperFunctions
```

First, we need to define some straightforward helper functions

```
expanded :: Node -> Bool
expanded (Nd _ _ _ _ [] [] [])  = True
expanded  _                                = False

badNode :: Node -> Bool
badNode (Nd _ _ _ _ [] x _) | not(null x) = True
                            | otherwise   = False
badNode _                                 = False

badTab :: Tableau -> Bool
badTab = any badNode

firstbad :: Tableau -> Node
firstbad [] = undefined
firstbad (n : ns)
  | not (badTab (n:ns)) = undefined
  | badNode n = n
  | otherwise = firstbad ns

removebad :: Tableau -> Tableau
removebad tab | badTab tab = tab \\ [firstbad tab]
              | otherwise = tab
```

Next is the function expand, which roughly corresponds to a "one step expansion" of a given tableau, and is given by

```
expand :: Tableau -> [Tableau]
expand tab | not (badTab tab) = [concatMap (head . step) tab]
           | otherwise = [ head (step (firstbad tab)) ++ removebad tab, last (step (
               firstbad tab)) ++ removebad tab ]
```

where a tableau is considered "bad" if it has at least one "bad" node, i.e. a node which has empty (i) and nonempty (ii)
In the first case, if the tableau in question is not a bad one, then the output of expand is a list containing a single tableau, which is obtained by expanding every node on the leaf level by one step.
In the second case, so if the tableau has at least one bad node n, the expand function returns two tableaux, each of them differing from the original by an application of the step rule on n according to the nondeterministic behaviour of step on bad nodes.

One then needs the additional helper function

```
pairify :: [Tableau] -> [(Tableau, Tableau)]
pairify = map \x-> (filter (not . expanded) x , filter expanded x)
```

which takes a list of tableaux and splits every tableau contained in it into its expanded and nonexpanded parts, yielding a list of pairs of tableaux. Note that, in accordance to the behaviour of our step function, expanded nodes are only stored if they correspond to open branches, therefore, as soon as a tableau has an expanded node, one can be certain that it will not develop into a closed tableau.

The two functions above are then incorporated in the recursive behaviour of solve, which is given by

```
solve :: [(Tableau, Tableau)] -> Bool
solve []        = False
solve (p:pairs) | null (snd p) && not (null (fst p)) = solve (pairify (expand(fst p))++
    pairs)
                | null (snd p) && null (fst p) = True
                | otherwise = solve pairs || False
```

The idea behind it is the following: given a formula φ one starts with a list containing just the pair (T, [ ]) where T corresponds to the initial tableau associated to φ (which is different depending on whether one wants to check intuitionistic or classical validity of φ, and is given by functions which are present in the code) and applies the recursion in solve.
The first calculation step will look like

$$\text{solve}[(\text{T}, [\,])] \implies \text{solve (pairify (expand(T))}$$

now, if expand(T) has expanded nodes (i.e. fully expanded open branches), then solve will go into its third case and output solve [ ] || False, which yields False. Otherwise, it will continue to operate recursively, generating potentially more than one tableau. The reason why solve takes a list of pairs of tableaux as opposed to a list of tableaux is that one wants to check whether a tableau has fully expanded open branches at each step of the recursion, so that the procedure of expanding a tableau can short-circuit in case it is certain that the tableau will never close. Since the step rule is designed so that a branch is cancelled as soon as it closes, a closing tableau is one that eventually becomes the empty list, therefore our solve function yields True only when given a pair of empty tableaux.

Note that, given the way the solve function is written, it will work "depth-first" in the following sense : if, during the "solving", a list of pairs of tableaux has been generated, the solve function will work by expanding the tableau corresponding to the first pair as much as possible until it

either yields a closing tableau (in which case the entire function stops and yields True) or it yields a non-closing tableau, in which case the solve function will turn to expanding the other tableaux in the list.

Lastly we need these for quick usability:

```
initintuitTab :: Frm -> Tableau
initintuitTab  f = [Nd [] [] [] [] [S (F, f)] [] []]

intuitthm :: Frm -> Bool
intuitthm f = solve [(initintuitTab f, [])]

intuitsat :: Frm -> Bool
intuitsat f = not ( intuitthm (N f))

initclassTab:: Frm -> Tableau
initclassTab  f = [Nd [] [] [] [] [S (Fc, f)] [] []]

classthm :: Frm -> Bool
classthm f = solve [(initclassTab f, [])]

classsat :: Frm -> Bool
classsat f = not (classthm (N f))
```

```
module MyQuickCheck where
import Formulas
import Tableau
import HelperFunctions
import Step
import Solve
import Test.QuickCheck
```

The project includes a method for generating random formulas, which are then used to test the prover. This is done as follows (these lines are essentially taken from lecture slides):

```
atoms :: [Int]
atoms = [1..9]
instance Arbitrary Frm where
  arbitrary = sized randomForm where
    randomForm :: Int -> Gen Frm
    randomForm 0 = P <$> elements atoms
    randomForm n = oneof [ P <$> elements atoms , N <$> randomForm (n `div` 2), C <$>
        randomForm (n `div` 2) <*> randomForm (n `div` 2), D <$> randomForm (n `div` 2) <*>
         randomForm (n `div` 2)  ]
```

(the dollar sign is not preceded by a backslash in the original code, here a backslash is added so that it is possible to show the code).

Once this is settled, one can define the property corresponding to Glivenko's theorem, which states that for every propositional formula φ,

$$\varphi \text{ is classically valid} \iff \neg\neg\varphi \text{ is intuitionistically valid}$$

This is done as

```
glivenko :: Frm -> Bool
glivenko f = classthm f == intuitthm (N (N f))
```

which then makes it possible to run the test the validity of Glivenko's theorem on 100 randomly generated propositional formulas.

In our attempts, the prover passed these tests, so for every formula φ generated, it has given the same answer to the two questions "is φ a theorem of classical logic?" and "is $\neg\neg\varphi$ a theorem of intuitionistic logic?".

3 Conclusions

The project includes a prover for the implication-free fragment of intuitionistic propositional logic as well as a prover for the implication free fragment of classical propositional logic. The impossibility of testing whether formulas containing implications are intuitionistically valid is an obvious shortcoming of the project, and it is the first thing that comes to mind when thinking about directions to expand this work. This should be possible without changing the structure of the program too much. On the other hand, it is well known that classical implication is definable in terms of the other connectives, therefore the classical prover can actually be used as a prover for all of classical propositional logic. Since this prover is obtained by a slight modification of an intuitionistic prover, our conjecture is that it is way less performant than most provers developed specifically for classical logic, whereas it is not known to us whether other existing provers for intuitionistic propositional logic perform much better than ours.

References

[1] Miglioli, Pierangelo, Ugo Moscato, and Mario Ornaghi. "An improved refutation system for intuitionistic predicate logic." *Journal of Automated Reasoning* 13, no. 3 (1994): 361-373.

[2] Ferrari, Mauro, Camillo Fiorentini, and Guido Fiorino. "A tableau calculus for propositional intuitionistic logic with a refined treatment of nested implications." *Journal of Applied Non-Classical Logics* 19, no. 2 (2009): 149-166.

[3] van Eijck, J. *First Order Theorem Proving*, University of Amsterdam, 2016 Available at

`https://staff.fnwi.uva.nl/d.j.n.vaneijck2/courses/16/fsa/`