Method of Analytic Tableaux In Haskell

Robert Gorrie – McMaster University – April 11, 2017

1 Introduction

The following document outlines in detail a Haskell implementation of the Method of Analytic Tableaux. The method in question is a technique and a proof procedure used for checking the validity of a propositional formula. Truthfully, the procedure is a general one, and applies to many different types of logics. The following implementation is, however, only defined over basic propositional logic.

This document is organized as follows. In the first section we offer a brief summary of the Method of Analytic Tableaux; for readers who already understand the procedure, we advise against skipping this section as some of the terminology used will be integral to forthcoming sections. In the following section we introduce our datatypes and functions for defining, building, and interfacing with propositional formulas. Lastly, in the remaining section we build our implementation of the Tableaux method from the ground up, while introducing a new collection interconnected functions.

This document assumes a beginner level of Haskell comprehension.

2 Tableaux Method in Brief

The method is rather simple, and we will attempt to make this as quick and painless as possible.

Abstractedly, the procedure works by decomposing a given formula using a set of rules. These decompositions produce a tree structure in which each leaf is a propositional symbol with an assignment of TRUE or FALSE.

Direct your attention to the figure immediately below. Each operator either splits the tree ($\vee$) or pairs child elements together ($\wedge$) and is affected by whether a NOT ($\neg$) is applied to the whole of the operation.

$$\begin{array}{|lcl}
\neg(A \wedge B) & = & \neg A \vee \neg B \\
\neg(A \vee B) & = & \neg A \wedge \neg B \\
\neg(\neg A) & = & A \\
\neg(A \Rightarrow B) & = & A \wedge \neg B \\
A \Rightarrow B & = & \neg A \vee B \\
A \Leftrightarrow B & = & (A \wedge B) \vee (\neg A \wedge \neg B) \\
\neg(A \Leftrightarrow B) & = & (A \wedge \neg B) \vee (\neg A \wedge B)
\end{array}$$

We will now turn to the following example which will (hopefully) clarify the procedure. In it we consider the propositional formula $(p \wedge q) \Rightarrow q$ which is a clear tautology.

$$\begin{tikzcd}
& & ((p \wedge q) \Rightarrow q) \arrow[dl] \arrow[dr] & \\
& \neg(p \wedge q) \arrow[dl] \arrow[dr] & & q \\
\neg p & & \neg q &
\end{tikzcd}$$

We first apply the 'Implies' rule, splitting the tree, then on the left side we apply the 'Not And' rule, splitting the tree again.

Let us denote a function t which takes a single formula as a parameter and outputs a list for every subtree.

Let $F = (p \wedge q) \Rightarrow q$
Taking $t(F)$ we get $[\neg p]$, $[\neg q]$, and $[q]$

The final step of the procedure is then to check the subtrees for contradictions. Because we only have splits ($\vee$) and no pairs ($\wedge$) we are left with the subtrees $[\neg p]$, $[\neg q]$, and $[q]$, each subtree is free of contradictions. If we started with $\neg(\_ \Rightarrow \_)$ (instead of $(\_ \Rightarrow \_)$), we would have resulted with $[p,q,\neg q]$ as our only subtree, which houses a contradiction.
If ALL of the produced subtrees possess a contradiction, then the NEGATION of the original formula is valid. Noting this we can confirm that $\neg F$ is valid.

3 Form Data Type

The following section and corresponding module details a data type for expressing propositional formulae. Much of this section is borrowed from or inspired by Dr. Wolfram Kahl's 'Expr' module.

```
module Form where
```

First we build a free type representing the possible binary operators over propositional formulae. An equivalence operator, 'Equiv', is included for usability's sake, and is not required.

```
data BinOp = Implies | Or | And | Equiv
```

A formula is then one of the following: a propositional symbol (denoted by a single Char), the negation of a formula, or the application of a binary operator applied to two subformula.

```
data Form = P Char
  | Not Form
  | Bin BinOp Form Form
```

3.1 Interfacing with Form

We can represent the following formulas using our type:

$$\neg p \wedge p$$
$$(p \wedge q) \Rightarrow q$$
$$\neg(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$$
$$\neg((A \Rightarrow B) \vee (B \Rightarrow A))$$

They are, in order, as follows:

```
f1, f2, f3, f4 :: Form
f1 = Bin And (Not (P 'p')) (P 'p')
f2 = Bin Implies (Bin And (P 'p') (P 'q')) (P 'q')
f3 = Not (Bin Equiv (Bin Implies (P 'A') (P 'B')) (Bin Implies (Not (P 'B')) (Not (P 'A'))))
f4 = Not (Bin Or (Bin Implies (P 'A') (P 'B')) (Bin Implies (P 'B') (P 'A')))
```

Note that f1, f3, and f4 are tautologies, and f2 is the same F mentioned in the previous section.

The following is a function which returns the string representation of the offered formula.

```
showForm :: Form → ShowS
showForm (P p) = (p:)
showForm (Not f) = ('~':) ∘ ('(':) ∘ showForm f ∘ (')':)
showForm (Bin op f1 f2) = ('(':) ∘ showForm f1 ∘ showOp op ∘ showForm f2 ∘ (')':)
showOp :: BinOp → ShowS
showOp Implies = (" => "++)
showOp Or = (" | "++)
showOp And = (" & "++)
showOp Equiv = (" == "++)
```

ShowS is used for efficiencies sake (a simple recursive concatination yields quadratic complexity). Because ShowS is itself a function from String to String, we provide showF which calls showForm with an empty list, yielding the appropriate String.

```
showF :: Form → String
showF f = showForm f []
```

You can test this module by first loading it in GHCI (":l Form.lhs") and then calling showF on one of f1, f2, f3, f4.
Additionally, you can define your own propositions using the constructor rules of Form ("let f = Not (Bin Or ...)", etc).

4 Analytic Tableaux Implementation

The following section and module document the implementation of the Method of Analytic Tableaux. This is done through the application of Haskell functions to elements of type Form.

```
module Tableaux where
import Form
```

The following ”Form Tree” type represents the tree structure that is given by applying the decomposition rules (section 2) to a form. We will need to define some function for taking Form to FTree.

```
data FTree = Pair FTree FTree | Split FTree FTree | Leaf (Char, Bool)
```

We will now begin introducing functions. We will start from the highest level of abstraction, and proceed to introduce functions as they are needed.

Our first function is simple. It evaluates the validity of a Form. We know that we need to compute an FTree given our input, which we will then evaluate to test for validity. We feed our input Form 'f' to some function genFTree, and feed the output to some function evalTree.

```
evalForm :: Form → Bool
evalForm f = (evalTree (genFTree f) [])
```

Don't be confused by the empty list applied to evalTree; it is a recursive parameter for the function and will be clarified shortly.

We now present genFTree, taking Form to FTree. genFTree works inductively over the structure of Form. There is a case for every possible occurance of a binary operator, as well as one for each operator when paired with a NOT. Each recursive case corresponds to a rule from figure 2.1. Recall splits (∨) and pairs (∧) as discussed previously.

```
genFTree :: Form → FTree
genFTree (P v) = Leaf (v, True)
genFTree (Not f) = case f of
  Not f′ → genFTree f′
  P v   → Leaf (v, False)
  Bin op f1 f2 → case op of
    And → Split (genFTree (Not f1)) (genFTree (Not f2))
    Or → Pair (genFTree (Not f1)) (genFTree (Not f2))
    Implies → Pair (genFTree f1) (genFTree (Not f2))
    Equiv → Split
      (Pair (genFTree (Not f1)) (genFTree f2))
      (Pair (genFTree f1) (genFTree (Not f2)))
genFTree (Bin op f1 f2) = case op of
  And → Pair (genFTree f1) (genFTree f2)
  Or → Split (genFTree f1) (genFTree f2)
  Implies → Split (genFTree (Not f1)) (genFTree f2)
  Equiv → Split
    (Pair (genFTree f1) (genFTree f2))
    (Pair (genFTree (Not f1)) (genFTree (Not f2)))
```

The following Branch datatype represents the return type of the function t mentioned in section 2. Note that it is a list of (Char,Bool). Each pair holds a propositional symbol with its evaluation as computed by genFTree. It takes a list of Branches to properly describe an entire FTree, as a Branch is equivalent to one subtree.

```
type Branch = [(Char, Bool)]
```

The upcoming (dense and girthy) function, evalTree, was what we conceptualized in evalForm when we said there would be some function to evaluate the validity of some Form based on its FTree. The function takes an FTree and returns a Bool (denoting validity).

This function is defined inductively over the structure of FTree. It works by adding propositional variables from the FTree into a Branch as it encounters them. This Branch is our second parameter (recall the empty list in evalForm).

Once the FTree has been followed all the way to its leaves, some function evalBranch is called to evaluate the built up Branch, which is where the return type Bool comes from. evalBranch, once we write it, will correspond to checking a Branch (t output) for contradictions.

When evalTree encounters a Pair ($\wedge$), it puts the two children into the same Branch. If it encounters a Split ($\vee$), it calls evalTree twice, signifying two seperate Branches. The one exception to this simple scheme lies wherein a Pair is encountered in which both of it's children are Pairs. When this happens we OR four evalTree instances, corresponding to all the possible combinations of children; if a contradiction is found between any of them then a contradiction is found in the original Pair.

```
$evalTree :: FTree \rightarrow Branch \rightarrow Bool$
$evalTree\ (Pair\ f\ t)\ b$
    $|\ (isLeaf\ f) \wedge (isLeaf\ t) = evalBranch\ ((peelLeaf\ f) : (peelLeaf\ t) : b)$
    $|\ (isLeaf\ f) = evalTree\ t\ ((peelLeaf\ f) : b)$
    $|\ (isLeaf\ t) = evalTree\ f\ ((peelLeaf\ t) : b)$
    $|\ (isPair\ f) \wedge (isPair\ t) = (evalTree\ (Pair\ (digLeft\ f)\ (digLeft\ t))\ b) \vee$
      $(evalTree\ (Pair\ (digLeft\ f)\ (digRight\ t))\ b) \vee$
      $(evalTree\ (Pair\ (digRight\ f)\ (digLeft\ t))\ b) \vee$
      $(evalTree\ (Pair\ (digRight\ f)\ (digRight\ t))\ b)$
    $|\ (isSplit\ f) \wedge (isSplit\ t) = (evalTree\ (Pair\ (digLeft\ f)\ (digLeft\ t))\ b) \wedge$
      $(evalTree\ (Pair\ (digLeft\ f)\ (digRight\ t))\ b) \wedge$
      $(evalTree\ (Pair\ (digRight\ f)\ (digLeft\ t))\ b) \wedge$
      $(evalTree\ (Pair\ (digRight\ f)\ (digRight\ t))\ b)$
    $|\ (isPair\ f) = (evalTree\ (Pair\ f\ (digLeft\ t))\ b) \wedge$
      $(evalTree\ (Pair\ f\ (digRight\ t))\ b)$
    $|\ otherwise = (evalTree\ (Pair\ t\ (digLeft\ f))\ b) \wedge$
      $(evalTree\ (Pair\ t\ (digRight\ f))\ b)$
$evalTree\ (Split\ f\ t)\ b$
    $|\ (isLeaf\ f) \wedge (isLeaf\ t) = evalBranch\ ((peelLeaf\ f) : b) \wedge$
      $evalBranch\ ((peelLeaf\ t) : b)$
    $|\ (isLeaf\ f) = (evalTree\ t\ b) \wedge (evalBranch\ ((peelLeaf\ f) : b))$
    $|\ (isLeaf\ t) = (evalTree\ f\ b) \wedge (evalBranch\ ((peelLeaf\ t) : b))$
    $|\ otherwise = (evalTree\ f\ b) \wedge (evalTree\ t\ b)$
$evalTree\ (Leaf\ (c, n))\ b = evalBranch\ ((c, n) : b)$
```

The functions digRight and digLeft are very simple and will be declared shortly. They return either the right or left child (respectively) of an FTree.

peelLeaf is simply used to access the (Char, Bool) once we reach it in the FTree.

And at last, we have reached the final piece of the puzzle. The upcoming function evalBranch, which is called from evalTree for each Branch it generates, takes as input a Branch, and returns a Bool signifying if it found a contradiction.

Because True is returned when a contradiction is found (p and $\neg p$ existing in the same Branch, for any p,) The original function evalForm returns TRUE if the NEGATION of the original formula is valid. This may seem counterintuitive, but changing these return values so deep in the problem results in a cluttering of evalTree's logic. If we wish, we can make adjustments at a higher level.

Stated formally,

$$(\forall \text{ f: Form (evalForm (f) = True} \iff \text{isValid}(\neg\text{f})))$$

```
evalBranch :: Branch → Bool
evalBranch [] = False
evalBranch (x : xs) = case lookup (fst x) xs of
  Nothing → evalBranch xs
  Just b → if b ≡ (snd x)
    then evalBranch xs
    else True
```

The following are definitions for each of the auxilliary functions used in evalTree. The dig functions as well as peelLeaf are defined for every instance of FTree simply for completion's sake, a dig will never be called for a Leaf and peelLeaf will never be called for a Split or a Pair (see evalTree for clarification).

```
digLeft :: FTree → FTree
digLeft (Split f t) = f
digLeft (Pair f t) = f
digLeft (Leaf l) = Leaf l

digRight :: FTree → FTree
digRight (Split f t) = t
digRight (Pair f t) = t
digRight (Leaf l) = Leaf l

peelLeaf :: FTree → (Char, Bool)
peelLeaf (Leaf l) = l

isPair :: FTree → Bool
isPair (Pair _ _) = True
isPair _ = False

isSplit :: FTree → Bool
isSplit (Split _ _) = True
isSplit _ = False

isLeaf :: FTree → Bool
isLeaf (Leaf _) = True
isLeaf _ = False
```

4.1 Interfacing with Tableaux

By this point we have constructed enough to (rather ruggedly) determine the validity of propositional forumlas. To do so, load the module in GHCI (":l Tableaux.lhs") and call evalForm for any of the previously defined Form elements (again, feel free to build your own using the Form constructors).

4.2 Extending to Satisfiability

The Tableaux method can be used to derive satisfiability as well as validity. The way this can be accomplished is rather straightforward. If every branch contains a contradiction, the negation of the formula is valid. If no branches contain a contradiction, the formula itself is satisfiable.

The first function we define in this section resolves the previously mentioned counterintuition of evalForm. It is very simple, and should be self explanatory.

```
isValid :: Form → Bool
isValid f = evalForm (Not f)
```

This next function, when fed a formula, will return True if it is satisfiable. By extension, a return value of False indicates that the formula is a contradiction.

```
isSat :: Form → Bool
isSat f = evalTreeSat (genFTree f) []
```

Note that this function is identical to our original evalForm, with the exception that evalTree has been replaced by some new function 'evalTreeSat'. This function will be very close to evalTree, with a few differences. We will also need to make adjustments to evalBranch.

There are two differences between evalTreeSat and evalTree. The first is that we now use evalBranchSat (defined below) instead of evalBranch. The second is that in the instance where we have a Pair followed by two Pairs we now combine all the cases with AND instead of OR. The reason for this is that in evalTree a contradiction in any of the subcases was enough to satisfy the requirements, but in evalTreeSat EVERY subcase must be free of contradictions.

```
evalTreeSat :: FTree → Branch → Bool
evalTreeSat (Pair f t) b
  | (isLeaf f) ∧ (isLeaf t) = evalBranchSat ((peelLeaf f) : (peelLeaf t) : b)
  | (isLeaf f) = evalTreeSat t ((peelLeaf f) : b)
  | (isLeaf t) = evalTreeSat f ((peelLeaf t) : b)
  | (isPair f) ∧ (isPair t) = (evalTreeSat (Pair (digLeft f) (digLeft t)) b) ∧
    (evalTreeSat (Pair (digLeft f) (digRight t)) b) ∧
    (evalTreeSat (Pair (digRight f) (digLeft t)) b) ∧
    (evalTreeSat (Pair (digRight f) (digRight t)) b)
  | (isSplit f) ∧ (isSplit t) = (evalTreeSat (Pair (digLeft f) (digLeft t)) b) ∧
    (evalTreeSat (Pair (digLeft f) (digRight t)) b) ∧
    (evalTreeSat (Pair (digRight f) (digLeft t)) b) ∧
    (evalTreeSat (Pair (digRight f) (digRight t)) b)
  | (isPair f) = (evalTreeSat (Pair f (digLeft t)) b) ∧ (evalTreeSat (Pair f (digRight t)) b)
  | otherwise = (evalTreeSat (Pair t (digLeft f)) b) ∧ (evalTreeSat (Pair t (digRight f)) b)
evalTreeSat (Split f t) b
  | (isLeaf f) ∧ (isLeaf t) = evalBranchSat ((peelLeaf f) : b) ∧ evalBranchSat ((peelLeaf t) : b)
  | (isLeaf f) = (evalTreeSat t b) ∧ (evalBranchSat ((peelLeaf f) : b))
  | (isLeaf t) = (evalTreeSat f b) ∧ (evalBranchSat ((peelLeaf t) : b))
  | otherwise = (evalTreeSat f b) ∧ (evalTreeSat t b)
evalTreeSat (Leaf (c, n)) b = evalBranchSat ((c, n) : b)
```

Lastly, you will find that the only difference in our new implementation of evalBranch is the replacement of False for True on the empty list pattern and the replacement of True for False when a contradiction is found. This will return True if there are no contradictions in the Branch. Paired with our changes to evalTreeSat, isSat can now determine whether a formula is satisfiable.

```
evalBranchSat :: Branch → Bool
evalBranchSat [] = True
evalBranchSat (x : xs) = case lookup (fst x) xs of
  Nothing → evalBranchSat xs
  Just b → if b ≡ (snd x)
    then evalBranchSat xs
    else False
```

We can now determine whether a propositional formula is valid, satisfiable, or invalid.

Enjoy.