Lecture 7 CS4860 Fall 2010

Completeness of the Tableau Proof System

The center piece of Smullyan Chapter II section 2 is his proof of the completeness theorem. It's Theorem 2 on page 28, only the second statement in his book that he labels as a theorem. Theorem 1 on page 26 is the basic result used to prove and motivate Theorem 2. Obviously he considers these to be very important statements since they are the only results in 26 pages full of technical statements that he calls "theorems." (In the hierarchy of labels used in mathematical writing, we see "remarks", "facts", "observations", "lemmas", "corollaries", "conjectures", and finally "theorems", "named theorems" and "theses" -- such as Church's Theorem and the Church/Turing Thesis, two statements at the top of this hierarchy.) Anyway, the completeness theorems in Smullyan are very important, both conceptually and in the history of the famous *Hilbert Program* that we mentioned in the Story of Logic. I think their statements and proofs are worth knowing in detail and hope that every student will be able to state them, explain them, and prove them before the course is over. Let me start with the completeness theorem, as Smullyan states it and then present it symbolically as well. First we recall a definition from class.

Definition: For X a formula, let Var(X) denote the subset of Var consisting of all propositional variables that occur in X.

Theorem (Completeness of Tableau): (a) If X is a tautology, then every completed tableau for FX must close, and (b) Every tautology is provable by the tableau method.

Theorem (Completeness): All X:Form.((All v:(Var(X) → **B**). val(v,X) = t) implies
(All T:Tableau(FX). (Completed(T) implies Closed(T))) &
(Exists T:CompleteTableau(FX).Closed(T))).

As we noted with the consistency theorem, the **proof plan** is 90% of the job of proving this result. Just as for the consistency theorem in the last lecture, the contrapositive formulation gives us a better approach to the proof because it matches our intuitions well, especially the intuition that a tableau proof is a failed attempt to *systematically find a counter example to* X, that is to **falsify** X.

So Smullyan proves *(not Provable implies not Tautology)* and states it in the form, *(not Closed implies Sat(FX))* or in other words *(Open implies Sat(FX)).* He states this as Theorem 1.

Theorem (Open Branch): Any complete open branch of any tableau T is simultaneously satisfiable.

Symbolically this is stated as follows:

Theorem: For all X:Form. All T:Tableau(FX). All p:Path(T).(Completed(p) & Open(p) implies Exists v:Var(X) → B. All Y:SForm(p). val(v,Y) = sign(Y) in **B**).

What we mean by SForm(p) is the set of (signed) formulas on the path p of T. We mean by sign(Y) the truth value corresponding to the sign of Y, so sign(FY) = f in **B** and sign(TY) = t in B.

Why is this result so intuitive clear to us? It's because we explained tableau proofs as systematic searches for a counter example, a falsifying assignment. The rules were designed to make sure we explored every possible way to falsify FY or make true a formula TY. Consider the two rules for implies. These are the conceptually hardest rules.

```
T(X imp Y)          F(X imp Y)
.  FX|TY                TX
.                       FY
```

By consulting the semantics for the connectives on page 15, called "facts" by Smullyan, we see that in items 4a and 4b he says this about the semantics.

4a. If (X imp Y) is true then either X is false or Y is true
4b. If (X imp Y) is false then X is true and Y is false

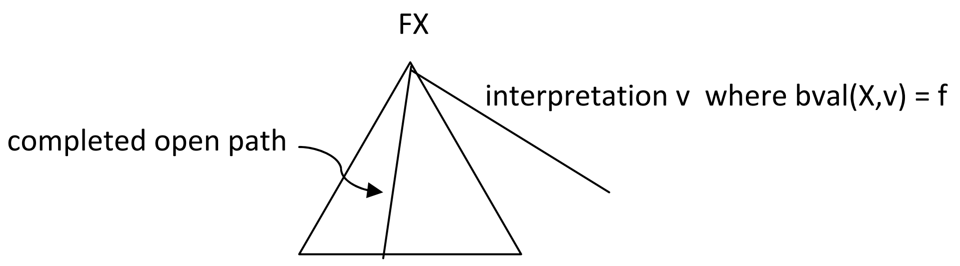
Instead of using *if then* he could say

4a. (X imp Y) is true iff either X is false or Y is true
4b. (X imp Y) is false iff X is true and Y is false

and then it would be even more clear that the two rules capture all the ways of making the implication true and making it false. This is the case for all rules, so the search for a falsification is not missing any possibilities, that is, **the search is complete**.

Proof

The proof method is to build the interpretation v from the completed open path p.



Smullyan notes that the finite set of formulas on the completed open path p satisfy these three properties:

H0: No variable receives both the sign T and the sign F on the path p.

H1: If an alpha formula appears on p, then so do both alpha1 and alpha2.

H2: If a beta formula appears on p, then either beta1 or beta2 appears on p.

It is easy to verify these properties. H0 is true because otherwise the path p would be closed, but we are given that it is open. The other two properties are immediate from the form of the rules.

These properties tell us that we can build interpretation v by giving the propositional variables the values of their signs and then all the formulas on the path, including the goal FX, get the value of their signs by the facts on page such as 4a and 4b etc. We prove this by induction on the degree of the signed formulas on the path p. See page 27.

Qed

Smullyan proves this result for a possibly infinite set S. Sets satisfying the H properties are called *Hintikka sets*. Smullyan shows that any Hintikka set (also called *saturated downwards*) is satisfiable by the method we used to prove that all formulas on the path p get their sign as value under v.

Remarks on Metamathematics

We spent a considerable amount of time analyzing Smullyan's description of Boolean evaluations, including writing a recursive function called *bval* that computes a Boolean evaluation function. The type we gave this function is

bval belongs to the function type $(\text{Var} \to \mathbf{B}) \to (\text{Form} \to \mathbf{B})$

This type is not actually what we use in computation. We don't expect to provide as input a valuation for all propositional variables when we only need the valuation on the variables that occur in the formula we are evaluating, say X. So recall that Var(X) for X a formula is the set of propositional variables that occur in X. Then the function we want belongs to this type:

$$X{:}\text{Form} \to (\text{Var}(X) \to \mathbf{B}) .$$

This is the exact right type for a Boolean valuation, but there is no standard programming language that allows this type, called a *dependent function type*. On the other hand, the type systems of several *interactive proof assistants* such as Agda, Coq, Nuprl, and MetaPRL do allow this type for their computable functions. So we will use it here for this new version of bval with the inputs in the right order for computation.

```
Bval(X,v) = case X is var(y) → v(y); neg(U) → bnot(bval(U,v));
op(U,V) → bop(bval(U,v),bval(V,v)) end
```

During the lecture we discussed the type Var → **B** and noted that it is a very “big type” compared to Form in the sense that Var → B is *uncountable* while Form is *countable*. But in fact, even when we consider this large type, we are only interested in the computable functions in that whole collection of functions. We will take a closer look at this type later when we examine compactness.