Lectures 4 and 5 **CS4860 Fall 2010**

In Lecture 3 we translated Smullyan's Chapter I section on trees into a more symbolic language and made the definitions and theorems explicit as well as more precise. These notes will carry out this process for his section on Formulas and Boolean Valuations.

Translating Mathematical Text in Symbolic Logic

Learning to translate natural language into symbolic logic is an interesting skill taught in many logic courses. (Translating back into natural language is a research topic in *natural language processing*.) We are learning that skill in a very realistic setting by showing how to translate informal mathematics into symbolic logic. We are also showing how to expose implicit knowledge and make it explicit. Rather than showing how to translate into symbolic logic statements made in the newspapers, which many textbooks demonstrate and which might be fun, is not as realistic as what we are doing because most newspaper discourse is not intended to be precise or intricate -- nor to convey mathematical arguments! On the other hand, in the "real world," a trained logician is often needed to make precise what a system designer is writing about his or her design or to make precise the comments in a critical piece of software so that formal methods tools such as model checkers and theorem provers could be brought to bear on checking the validity of the design against specifications or verifying that assertions about code are true, perhaps exposing the programmers reasoning.

Moreover, this translation and explication exercise serves a dual purpose for us because in the process we come to understand the informal concepts much better and to uncover implicit assumptions or discover other options in stating the definitions and theorems, options which might be more accessible to some readers than what Smullyan wrote – even though his explanations are exemplary in many cases.

Let us start with Smullyan's definition of a formula on page 7, the one he and I both strongly prefer over the syntactic approach that he outlines on pages 5 and 6. The definition below is almost word for word a translation of the sentence "Under this plan, a formula is either a (propositional) variable, and ordered pair (if it is a negation) or an ordered triple."

Definition 1 A *formula* is either a (propositional) variable, or an ordered pair <~,X> (if it is a negation of formula X), or an ordered triple <X,op,Y> where X and Y are formulas and op is one of the three binary operators, *and, or, implies*.

Let Form be the type of all formulas over the type Var of propositional variables. We could show the dependence on Var by writing Form(Var) which would allow us to change the definition of the propositional variables.

The above definition of Form is *recursive* because we need to say that X and Y are formulas, the very concept we are defining. Why does this make sense? Why is this not a *circular definition* of the kind that worried Russell so much? It is not circular because we have in mind a finite process of "unfolding" the definition until we come only to the case of variables. But this is not so explicit in this definition. We can imagine carrying out the unfolding process "forever" in the sense that we just keep replacing the variable X and Y with other terms involving X and Y.

The way we say that we intend the finite unrolling is to mark the definition as recursive! Smullyan says this on page 5: "The notion of formula is given by the following recursive rules, which enable us to obtain new formulas from those already constructed." He then states his conditions F0 to F4, where the parentheses are part of the language.

We can give a definition that makes the unrolling process more explicit. This is sometimes called an *inductive definition.* Let S be a collection of objects and define the operation F which extends it by forming the following ordered pairs and triples {<~,X>, <X,op,Y> | X, Y in S}, that is

$$F(S) = \{<\sim,X>, <X,op,Y> \mid X, Y \text{ in } S\}$$

Definition 2: Let F(0) = Var and F(n+1) = F(F(n)), and let Form be Union F(i) for all natural numbers i.

With this definition it is clear that we are building only finite formulas, and we can locate exactly the level of n at which one of the formulas given by Def 1 is constructed in the type Form. However, there is a subtle point about this definition too. Normally when we define a function, we know in advance the type of its domain and range, so we declare F:A → B. In this case, the simplest type for F is Set → Set, and we know that there are issues with taking the set of all sets to be a set or a type. Later in the course I will show how this problem can be overcome in the case of types, and we can view F as a function on types. The key property that makes this definition work is that F is *monotone*, that is if S is a subset of S', then F(S) is a subset of F(S').

We can write another version of this type if we use the labeled disjoint union of two types. Let l:A + r:B be the type whose elements are all pairs <l,a> for a in A and <r,b> for b in B. The labels are l and r for "left" and "right" members of the pair.

Definition Form = var:Var + neg:Form + and:Form × Form + or:Form × Form + imp:Form × Form

We can build a *case discriminator* for elements of this type that assigns a value to an element depending on its structure. We write for X in Form

case X is

if var(v) then exp1(v)

if neg(U) then exp2(U)

if and(U,V) then exp3(U,V)

if or(U,V) then exp4(U,V)

if imp(U,V) then exp5(U,V)

end

To save space we sometimes write

case X var(v) → exp(v); neg(U) → exp2(U); and(U,V) → exp3(U,V);
or(U,V) →exp4(U,V); imp(U,V); imp(U,V) → exp5(U,V) **end**

We can simplify the case analysis further by grouping together all of the binary operators when we perform the same operation in each case. For example, here is how we can define the degree of a formula.

degree(X) = **case** X var(v) → 0; neg(U) → deg(U)+1; op(U,V) → deg(U) + deg(V) +1 **end**

Relationship to Trees

Each of these variants of Smullyan's definition of formulas has the property that the individual formulas have the structure of a tree. So why does he not define them as trees?

Exercise: Find the small discrepancy in treating formulas as trees in the way Smullyan defines them. (Hint, when he wants to refer to them as trees, as in formation trees on page 9, he uses the phrase "occurrence of" as in clause (i) Each end point is (an occurrence of) a propositional variable.

Induction Principles for Formulas

Smullyan provides an induction principle on page 8 based on the degree of a formula where

deg(pi) = 0, deg(<~,X>) = deg(X)+1, deg(<X,op,Y>) = deg(X)+deg(Y)+1.

Another simple induction principle is simply on structure. Here is the form of that principle.

To prove All X:Form.P(X), prove the base case where X is a variable, and the induction case. In the induction case, assume that P(Y) hold immediate subformulas of X and prove P(X).

Another more elegant induction principle is

Complete Structural Induction

To prove All X: Form. P(X), assume P(Y) holds for all subformulas of X and show that P(X) follows from this assumption. We can state the whole principle symbolically as follows.

All X:Form.(All Y:SubForm(X).P(Y) implies P(X)) implies All X:Fom.P(X)

Exercises: Prove using each of the tree induction principles the following facts about formulas.

1. For every formula X, the type of all subformulas of X, SubForm(X), is finite.
2. Given any formula X, if we replace every occurrence of *and* by *or*, the result remains a formula.
3. Given any formulas X and Y, if we replace every variable v in X by the formula Y, the result is a formula.
4. Given any formula X, if we replace every *and* by an *or*, the result is a formula.

Valuations and Boolean Valuations

We write some of Smullyan's definitions symbolically. He says that if S is any set of formulas, then a *valuation* v is a mapping of S onto the Booleans. We can say this symbolically by declaring that v has the type S → **B.**

He then defines the key concept of a *Boolean valuation* on page 10 using four conditions, B1 to B4. For example, in B1 he says that *not* X receives the value t if X receives the value f and it receives the value f if X receives the value t. We can present this symbolically. First we note that for each of the Boolean connectives, *not, and, or, imp,* there is a corresponding operation on the Booleans which we denote by *bnot*, *band*, *bor, bimp*. These are functions on Booleans, e.g.

bnot: **B** → **B** where bnot(t) = f and bnot(f) = t. Note bimp: **B** × **B** → **B** has the value f exactly on bimp(t,f). In general, if *op* is a Boolean operator, then let *bop* be the corresponding definition of the operator as given by the truth table for the Boolean operators.

Definition f: S → B is a *Boolean evaluation*, Boolean(f,S), on a set S of formulas iff for each X in S, f(X) = **case** X is var(v) → f(v); neg(U) → *bnot*(f(U)); and(U,V) → *band*(f(U),f(V)); or(U,V) → *bor*(f(U,V)); imp(U,V) → *bimp*(f(U),f(V)) **end**

We can simplify this using our convention on bop to get

case X is var(v) → f(v); neg(U) → *bnot*(f(U)); op(U,V) → bop(f(U) **end**

We can also use the case operator to define a function on Form that tells what case a formula satisfies. So for any X in Form, there are functions such as isneg(X) that has value t iff X is a negation. Here is the isneg function of type Form → B

isneg(X) **= case** X is var(v) → f; neg(U) → t; bop(U,V) →f **end**

In the same way we can define isvar, isand, isor, isimp from Form to **B**.

We also define functions that decompose a compound formula. Let

neg:{X:Form | isneg(X)} → Form &
All X:Form. isneg(X) implies X = <~,neg(X)>

andl, andr: {X:Form| isand(X) → Form &
All X. Form. isand(X) implies X = <andl(X), and, andr(X)>

orl,orr:{X:Form | isor(X) } → Form &
All X:Form. isor(X) implies X = <orl(X), or, orr(X)>

impl, impr:{X:Form | isimp(X)} → Form &
All X:Form. isimp(X) implies X = <impl(X), imp, impr(X)>

At the bottom of page 10 Smullyan states the following theorem and says it is “easily verified by induction on the degree of X.”

Theorem (p.10 Unique Boolean Evaluation): A Boolean valuation of the variables of any formula X can be extended to at most one Boolean valuation of the subformulas of X.

Proof: Suppose there are two Boolean valuations, v and w of the subformulas of X, and they agree on the variables of X, say that w extends v. For a proof by induction that All X:SubForm(X). v(X) = w(X) in **B**, assume that v and w agree on all immediate subformulas of X. Now it will be clear by cases on the structure of X that they must agree on X as well. For example, if isneg(X), then since v(Y) = w(Y), it is clear from condition B1 (the case of neg in the case operator) that v(X) = w(X). The same idea works for all cases. **Qed**

The more interesting question is whether there is even one Boolean valuation of Form. Smullyan proves this next, but his proof does not lead us to the standard way of building the valuation which I will provide after the proof. The informal statement is simply "Every interpretation v of the variables Var can be extended to a Boolean valuation of Form."

Theorem (p. 11 existence of a Boolean Valuation) For all v:Var → **B**. Exists f: Form → **B**. Boolean(f,Form) & For all x:Var. f(x) = v(x) in **B**.

Smullyan proves this using the following key lemma.

Lemma: For all v:Var → **B**. For all X:Form. Exists f: SubForm(X) → **B**. Boolean(f,SubForm(X)).

Proof: By induction on the structure of X: If isvar(X) then define f(X) = v(X). Otherwise, assume for induction that we have Boolean valuation functions f_Y defined for all Y in SubForm(X) and consider the structure of X.

1. If isneg(X) then define f(X) = bnot(f_Y(Y)).
2. If isbop(X), then let X = <Y1,op,Y2> and define f(X) = bop(f_Y1(Y1),f_Y2(Y2)))

Since Boolean(f_Y1,SubForm(X)) and Boolean(f_Y2,SubForm(X)), and those functions f_Y1, f_Y2 are unique, it is clear that the definition of f determines a specific function (that is f is "well defined") and that Boolean(f,SubForm(X)).

Qed

Since we proved the lemma for all X in Form, it is clear that the existence theorem follows immediately from the lemma by noting that X is a subformula of itself, that is, X belongs to SubForm(X).

For any formula X, and interpretation of the variables of X can be extended to at most one Boolean Valuation of the subformulas of X.

Smullyan must be pleased with his treatment of Boolean Valuations because he mentions it in the Preface to the book, page VII, saying "We use the term "Boolean valuation" to mean any assignment of truth values to all formulas which satisfies the usual truth-table conditions for the logical connectives. Given an assignment of truth –values to all propositional *variables*, the truth-values of all other formulas under this assignment are usually defined by an *inductive* procedure. We indicate in Chapter I how this inductive definition can be made explicit – to this end we find useful the notion of a *formation tree* (which we discuss earlier)."

The explicit method he has in mind is the remark in square brackets on page 11: [We might think of the situation as first constructing a formation tree for X, then assigning truth values to the end points in accord with the interpretation v_0, and then working our way up the tree, successively assigning truth values to the junction and simple points in terms of the truth values already assigned to their successors, in accordance with the truth table rules.]

Oddly enough, the standard recursive procedure can be thought of in this way. Let's write down that recursive procedure. Indeed it is the recursive process we see in the Lemma. The function f "calls" the functions f_Y1, and f_Y2 which become part of f by its definition. *So f as defined actually calls itself recursively.* We make this recursion explicit in the following definition.

Definition: Recursive Boolean valuation given interpretation v of Var

bval(v,X) = **case** X is var(x) → v(x); neg(U) → bnot(bval(U));
op(U,V) →bop(bval(U),bval(V)) **end**

Once we have this function in mind, we can state a stronger theorem about Boolean valuations.

Theorem (Strong Boolean Valuation): There is a unique function f whose (two) inputs are v, an assignment of truth values to Var, and a formula X such that f on its first input produces a Boolean Valuation of Form, i.e. Boolean(f(v), Form).

Symbolically this theorem is:

Exists! f: (Var → **B**) → ((Form → **B**) → **B**).
All v:Var → **B**. Boolean(f(v),Form)