Logic of Programs	**Lecture 2**
CS 6860 Fall 2015	**Thurs. Aug 27, 2015**

Lecture 2

Topics

1. Recap planned course outline.
2. Frege's Approach
   - Define conceptual content.
   - Find conceptual content (for constructive logic, this is *computational content*).
3. FOL as a programming language.
4. Smullyan presentation of Propositional Calculus (handout attached).
   - Syntax
   - Semantics
5. Examination in type theory of Smullyan's presentation of Propositional Logic.

   5a. Syntax – inductive definitions done formally, inductive definitions in type theory (as in Coq, Nuprl).

   Propositional Logic à la Smullyan

   (a) Four symbols: $\sim$, &, $\vee$, $\Rightarrow$.

   (b) Denumerable list of symbols P_1, P_2,.. call prop variables.

   (c) (,)

   Definition: A formula is defined *recursively* (inductively).

   F_0: every prop variable is a formula

   F_1: if A is a formula, so is $\sim A$.

   F_2, F_3, F_4: if A, B are formulas, so are $(A\&B)$, $(A \vee B)$, $(A \Rightarrow B)$.

   Proposition: "Every formula can be formed in only one way", Smullyan p.6.

   Inductive definition of Smullyan's formulas using type theory

   One way to define formulas formally, say using Nuprl, is to let PropVar be the type of Atoms or Tokens. We could even take them to be numbers. Then we can capture the formulas as the recursive type

   Form = PropVar + (Form + (Form × Form + (Form × Form + Form × Form)))

The type $A + B$ is called the disjoint union of A and B. Its elements have the form $\text{inl}(a)$ for $a \in A$ or $\text{inr}(b)$ for $b \in B$. The term inr stands for inject_right and inl for inject_left. To build a propositional variable, we inject a PropVar into this type. Let PropVar=$\{P_1, P_2, P_3, ...\}$, then $\text{inl}(P_1)$ is a formula.

By injecting a formula into the Form component, say $\text{inl}(\text{inr}(P_1))$, we can represent $\sim P_1$. To represent $P_1 \& P_2$ we inject the pair $< P_1, P_2 >$ into the first pair component, Form $\times$ Form, e.g. $\text{inl}(\text{inl}(< P_1, P_2 >))$. We represent $P_1 \vee P_2$ by injecting $< P_1, P_2 >$ into the second pair component, $\text{inl}(\text{inl}(\text{inr}(< P_1, P_2 >)))$.

This is an awkward way to designate formulas, but we could make up *display forms* to make them readable, e.g.

$\sim A$	displays	$\text{inl}(\text{inr}(A))$
$A \& B$	displays	$\text{inl}(\text{inl}(< A, B >))$
$A \vee B$	displays	$\text{inl}(\text{inl}(\text{inr}(< A, B >)))$.

A better recursive type is this:

Let Uop = $\{\sim\}$ Biop=$\{\&, \vee, \Rightarrow\}$

Form = PropVar $\cup$ (Uop $\times$ Form) $\cup$ (Biop $\times$ (Form $\times$ Form))

Now	$\sim A$	is	$<\sim, A>$
	$A \& B$	is	$<\&, < A, B >>$
	$A \vee B$	is	$<\vee, < A, B >>$
	$A \Rightarrow B$	is	$<\Rightarrow, < A, B >>$.
	$(A \& B) \Rightarrow A$	is	$<\Rightarrow, <\&, < A, B >>, A>$.

Nuprl allows this simple *union type*, $A \cup B$. Coq does not. The union type is subtle as we will see when we define the types more formally. Here it "works like a charm".

5b. Semantics – give an alternative *conceptual content* by providing computational content.

Smullyan gives the traditional Boolean semantics for the Propositional Calculus. We can derive this semantics, but we provide a more basic semantics based on computational evidence. We need to be clear about Frege's notion of conceptual content. So we need *judgement* – A.

In Nuprl and Coq these judgements are integrated into the combined judgement $H \vdash A$ where H is a list of formulas and A is a single formula.

5c. Proof rules – for &, $\Rightarrow$

Consider the judgement $\vdash (A \& B) \Rightarrow A$.

We will first look at it as a goal to prove. Implicit in this goal is that we know it *makes sense*, i.e. has conceptual content. We will establish this simultaneously with proving the goal. The proof will actually build the computational content as it progresses.

The computational meaning of $(A \& B) \Rightarrow A$ is that we can construct a computable

function f taking evidence for the proposition $A\&B$ and producing evidence for A. Since this must be a function, we use the lambda notation as the skeleton of the evidence.

$$\begin{array}{lll} & \vdash (A\&B) \Rightarrow A & \text{by } \lambda(h.\_\_). \\ h:(A\&B) & \vdash A & \text{by spread}(h;\, x, y.\_\_) \\ x:A,\, y:B & \vdash A & \text{by } x \end{array}$$

The computational content has been constructed by the proof. It is the functionn in lambda notation $\lambda(h.\text{spread}(h; x, y.x))$. Notice that this is an *untyped function expression*. Nevertheless, we can see from the derivation that it has type $A \times B \Rightarrow A$. It is easy to see that this is a "program".

The typing judgements are revealing the sense of the proposition.

Our way of knowing that this type is sensible come from the claim that $(A\&B) \Rightarrow A$ is a proposition. We have implicitly established this by simultaneously proving

$$\begin{array}{rl} (A\&B) \Rightarrow A & \text{in Prop} \\ A\&B & \text{in Prop} \\ A & \text{in Prop} \\ B & \text{in Prop} \end{array}$$