Topics

1. Questions or comments about the proof assistant demo.

2. Ideas for projects
   (a) read seminal papers, these approach the course topics from several points of view.
      • Per Martin-Löf: philosophy of knowledge
      • Bates & Constable: CS and programming
      • Bishop: real analysis and foundations of mathematics
      • Beeson: constructive Euclidean geometry
      • Jackson: automata theory in Nuprl
      • Cohen: ancestral logic (author is Cornell postdoc now)
      • Brouwer: the Fan Theorem and König’s lemma
   (b) use the Coq proof assistant on a course related topic
   (c) “implement” a logic of programs for OCaml or Haskell or...

3. Challenge Problem: is $\sim (\alpha \ast \beta) \rightarrow \sim \alpha \lor \sim \beta$ programmable?

4. Observation about “where we stand” on making our programming logic rigorous:
   (a) We do not have consistency and completeness theorems, we don’t even know what they mean yet!
(b) We seem to need these type constructors and their associated “code”:

\[
\begin{align*}
\& \text{ as } & \alpha \ast & \beta & \text{ with constructor } & \text{pair}(a; b) \text{ and } \\
& \text{ destructor } & \text{spread}(p;a,b,\_\_)
\end{align*}
\]

\text{implies as } \alpha \rightarrow \beta \text{ with constructor } \lambda(x.b(x)) \text{ and destructor } \text{ap}(f;a)

\text{or as } \alpha \lor \beta \text{ with constructor } \text{inl}(a), \text{inr}(b) \text{ and destructor } \text{decide}(d;a,\_\_;b,\_\_)

\text{false as void no constructor and destructor } \_

\text{true as unit constructor } \cdot

(c) These types seem to be essential whereas in Boolean logic we can use \&, \lor, \Rightarrow, \sim OR we can just have one connective - e.g. Sheffer stroke (Smullyan page 14, Exercise 5.)

(d) For the programming logic we need \textit{computation rules}, they will be critical in attempting to understand consistency and completeness. \textit{These rules will be part of the logic.}

- spread(pair(p_1,p_2); x, y.exp(x,y)) reduces to exp(p_1,p_2), that is, we substitute p_1 for x and p_2 for y in exp(x,y).

- ap(\lambda(x.b(x));a) reduces to b(a), i.e. a substituted for x in b(x)

- decide(inl(t_1);l.left(l); r.right(r)) reduces to left(t_1) 
  decide(inr(t_2);l.left(l); r.right(r)) reduces to right(t_2)

- any(e) is irreducible. (Note, it is interesting to think of any(e) as an exception where the expression e shows why this case is impossible.)

(e) Starting Smullyan Part II First-Order Logic p.43

i. This theory goes by several names. Here is a table of names and the authors of logic books that use these names.

<table>
<thead>
<tr>
<th>First-Order Logic</th>
<th>Predicate Logic</th>
<th>Functional Calculus</th>
<th>Quantification Theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smullyan</td>
<td>Predicate Calculus</td>
<td>Church</td>
<td>Mendelson</td>
</tr>
<tr>
<td>Boolos &amp; Jeffrey</td>
<td>Kleene</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bell &amp; Machover</td>
<td>Thompson</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoenfield</td>
<td>O’Donnell &amp; Constable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gallier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harrison</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ii. From the programming language point of view, these names are not explanatory. We want to talk about the type system. The key idea in this approach is dependent types. A dependent type is a type that depends on a value. The supplementary textbook by Simon Thompson, *Type Theory and Function Programming*, does a good job explaining this idea. The on-line sources are not very good except for the Wiki page, which gives a eurocentric view of this idea except when they mention Curry and Howard, two Americans who made major contributions along with de Bruijn and Scott. The Automath system of de Bruijn was the first to implement this idea, in 1970.

The dependent types connect programming logics directly to First-Order logic. So this topic will be treated here in an integrated way. The most transparent example of a dependent type might be the generalization of the polymorphic function type, $\alpha \rightarrow \beta$. We will allow the type $\beta$ to depend on values from the type $\alpha$. A simple syntax for this is $x : \alpha \rightarrow \beta(x)$. We allow the polymorphic type $\beta$ to depend on the type $\alpha$.

A concrete example of a dependent function type is to take $\alpha$ to be the integers, $\text{int}$, and to take $\beta(n) = x : \text{int} \times \{x \leq n\}$ then the type $n : \text{int} \rightarrow x : \text{int} \times \{x \leq n\}$ describes these functions that take an integer as input and produce a pair of an integer and a value in the type $\{x \leq n\}$ which we take to be empty if $x > n$ and to be $\text{unit}$ if $x \leq n$, i.e. $if \ x \leq n \ then \ \text{unit} \ else \ \text{void}$. There is a value in this type if and only if $x \leq n$. 
