Lecture 8

CS4860

Tuesday, September 20, 2016

1 Introduction

- New challenges and hints for old ones.
- Clearing up some misunderstandings.

No one yet has proved $\sim\sim(\alpha \vee \sim\alpha)$. But using the Refinement Logic, you can at least get a start.

Here is a Refinement Logic proof of an example from a student.

$\vdash ((\alpha \vee \sim\alpha) * (\alpha \Rightarrow \beta))$ λ(ad. ____)

ad : $(\alpha \vee \sim\alpha) * (\alpha \Rightarrow \beta)$ $\vdash \sim \alpha \vee \beta$ spread(ad; d; i. ____)

d: $(\alpha \vee \sim\alpha)$, i: $(\alpha \Rightarrow \beta)$ $\vdash \sim \alpha \vee \beta$ decide(d; a ____; na. ____)

a: α, i: $(\alpha \Rightarrow \beta)$ $\vdash \sim \alpha \vee \beta$ ap(i; ____; z.____)

$\vdash \alpha$ by a

z: β $\vdash \sim\alpha \vee \beta$ inr(z) z=ap(i;a)

na: $\sim\alpha$ $\vdash$ inl(na)

λ(ad. spread(ad ; d. decide(d; inr(ap(i;a));na. inl(na))))

Here is another example from another student:

$(p \vee \sim p) \rightarrow \sim\sim(p \vee \sim p)$

Everyone try $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow (\alpha * \beta \rightarrow \gamma)$ and
$(\alpha * \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow (\beta \rightarrow \gamma))$.

2 Topics

1. Review consistency and completeness for propositional calculus (prop. Boolean logic), recall the challenge of defining a logic of polymorphic programs or "polymorphic propositional logic". (Review of Chapter 2.)

2. In Chapter III, Smullyan extends logical results to "infinite sets" of Boolean formulas. He does it as preparation for First-Order Logic. He studies König's Lemma. Results of mine and my colleague of 22 years Mark Bickford cast a new light on these results, showing how to extend them to solve a long standing open problem about First-Order Logic. We will thus eventually extend Smullyan's results, especially his treatment of König's Lemma. (A former CS4860 students, Crystal Cheung, wrote an interesting MEng. thesis on this topic which we will discuss.)

3. This extension (to "infinite sets") will cause us to review Smullyan's definition of trees. We need to "program" his definitions and results. This is clarifying and introduces a theme of the course - we can "program" most theorems in this course in modern programming languages, and we can formalize *and prove* all of them in Agda, Coq, and Nuprl.

4. We are aiming to use these ideas to help us make precise the polymorphic programming logic that we have been discussing since Lecture 1.

   I proposed above two challenge problems about this task.

   - note that $\sim\sim(\alpha \vee \sim\alpha)$ was just solved.
   - $((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha)$ remains unsolved.

3 Review of Smullyan Chapter II

- Consistency (Soundness) Theorem (p.25)
  Provable by Tableaux implies tautology

- Completeness
  Tautology $\Rightarrow$ Provable
  (Theorem 2, p.28, includes 3 pages of proof and discussion.)

These two theorems have interesting "computational content." What is it? We say that the theorems have *constructive proofs*. We will gradually see in detail what this means.

4 Introduction to Smullyan Chapter III

Sketch of Smullyan Chapter III on compactness.

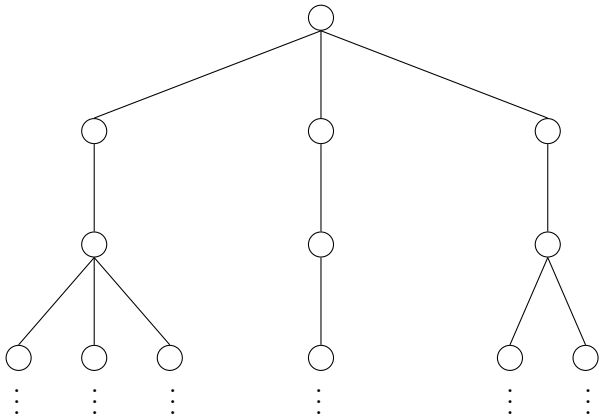
- What is he talking about and why is he doing it?

- Compactness Theorem

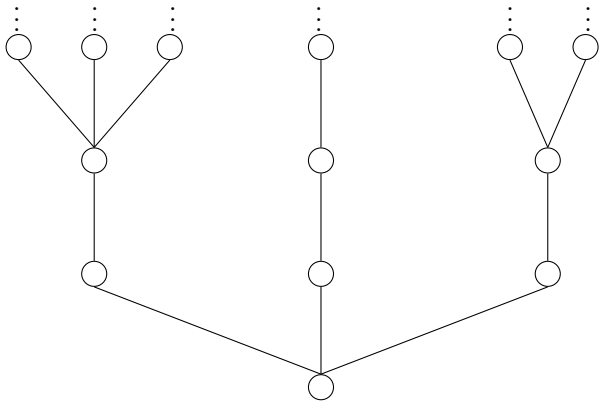
5 Review of Smullyan's Definition of Trees

5.1 Trees

Smullyan draws his trees with the root at the top, as in proofs.



In some literature that we will read, the trees *grow upward*. Growing upward is often a better way to visualize an unbounded tree.



5.2 Definitions

(Smullyan p.4)
A tree is *finitely generated* if and only if each point has only finitely many successors.

Smullyan uses this same definition on p.31, but it is ambiguous. Do we mean

1. $\forall$x:Points. $\exists$n: $\mathbb{N}$. | successor(x) | $<$ n

   OR

2. $\exists$n: $\mathbb{N}$. $\forall$x:Points. | successors(x) | $<$ n ?

We will say that in case (2) we have an n-ary tree. If n =2 we call the tree dyadic (p.4), if n=3, triadic, and so forth.

Smullyan p.3 defines an *ordered tree* by using a function Θ which for every junction point z assigns a co-list of successors. The list orders the successors, of which there could be an *unbounded number*, but ordered.