Lecture 38

Topics

1. PS5 None of the problems are difficult. We discussed problem 1 in Lecture 37 (assuming most students had read it). It seems unnecessary to discuss Turing machines in lecture beyond a few remarks. We did discuss Russell's paradox briefly, and it's in Wikipedia.
2. The propositions-as-types idea will be with us in all the rest of the lectures and on the final exam. It is in problem 5 and to some extent problem 1 and 2 as well. You could make it a feature of problem 2, say by proving that if a TM (Turing Machine) decides membership in a language Σ^* then there is a CTT function that decides. To impress us, you might want to show that *accepting* a language over Σ^* can be defined using partial types.
3. An issue that we face in this course – formal solutions and presentations from *Software Foundations* in Coq vs informal accounts in lecture. I don't believe in lecturing formally in a graduate course. You can get that from the books and Coq. You need to see options, alternatives, new approaches, and open issues and problems, in addition to formal material. This will become a social issue when we are cooperating more with machines.

Type theory required for PS5

We are not presenting the entire type theory of Nuprl, CTT. We are covering enough of the basic ideas so that you can use them in Problem Set 5. We are not requiring that you do any completely formal proofs, so we are not presenting all the rules, and none of them in detail. If you choose to do Problem 5 in Coq, Abhishek has sent information on Piazza, and we want you to explain the Coq extract and its relationship to the formal proof.

You will need a rule for induction on lists. The Nuprl rule is a form of "primitive recursion on lists". The form of an A list in CTT is given inductively. We say that *nil* is a list, and if t is a list, then for $h \in A$, $h.t$ is a list, with *head* h and tail t.

The rule for list induction is based on this computation rule.

$list\_ind(nil; base; u, v, w.body(u, v, w)) \downarrow base$

$list\_ind(h.t; base; u, v, w.body(u, v, w)) \downarrow body(h, t, list\_ind(t; base; u, v, w.body))$

This is a form of primitive recursion on lists. Thompson explains this well, and you can use his notation if you choose, or Coq notation.

For induction on $\mathbb{Z}$, the form is close to induction on $\mathbb{N}$ but there is a downward case as well. You will not need that rule for PS5. You can read the rule in the list of Nuprl rules written by Kreitz – posted with lecture 35.

Main topic of the next two lectures:

Math/Logic $\longleftrightarrow$ Computer Science
Correspondence
via
propositions-as-types

Here is what we've covered so far:

	Propositions	Types
1.	$A \Rightarrow B$ (implication)	$A \to B$ (function type)
2.	$\forall x : D.B(x)$ (universal quantification)	$x : D \to B(x)$ (dependent function type)
3.	$A \,\&\, B$ (conjunction)	$A \times B$ (Cartesian product)
4.	$\exists x : D.B(x)$ (existential quantifier)	$x : A \times B(x)$ (dependent Cartesian product)
5.	$A \vee B$ (disjunction)	$A + B$ (disjoint union) $\left\{ \begin{array}{l} x_1 : A_1 \mid x_2 : A_2 \mid ... \mid x_n : A_n \\ \text{(variants)} \end{array} \right\}$
6.	$False$	$Void$
7.	$\sim A$ (negation) $(A \Rightarrow False)$	$A \to Void$
8.	$a = b$ in A (equality) $axiom \in (a = a \text{ in } A)$	$a = b$ in A (equality) $(a = a \text{ in } A)$ is true

This correspondence between propositions and types is very deep and useful and evolving as new types are created for programming and new computational primitives are created with the type. We can use it both ways, seeing that a program or asserted program gives us the idea for a proof of a proposition. Conversely, a proof in a constructive logic can give us a provably correct program. Types give us rich ways to *specify a programming task*.

The correspondence has explanatory power. We can explain "classical logic" in terms of *computational evidence* rather than the less clear and tractable idea of *truth*.

The correspondence provides clear computational menaing to ideas such as "modular arithmetic" and other algebraic ideas. There are some types we will not have time to study deeply, such as intersection, $A \cap B$, and dependent intersection, $x : A \cap B(x)$, that introduce *new*

logical ideas, such as uniformity and *polymorphic truth*. This type also leads to a definition of *dependent records* and *objects*, e.g $\{x_1 : A_1;\ x_2 : A_2(x_1);\ x_3 : A_3(x_1, x_2), ...\}$.

This is a way to define *algebraic structures* such as groups, rings, fields, etc.

$\{D : Type;\ op : D \times D \to D;\ assoc : \forall x, y, z : D.op(x, op(y, z)) = op(op(x, y), z) \text{ in } D\}$

Programming and proving

We will explore the tight connection between programming and proving that is revealed through the propositions-as-types principle, supported in CTT, implemented om Agda, Coq, and Nuprl. At Cornell we were the first to make this real, implementing the λ-PRL system in 1980 and Nuprl in 1984. Also the 1978 PLCV programming logic integrated proofs and programs together in one system, based on constructive logic.

Consider the problem of finding the integer square root of a natural number. How could we do this? What is a good specification? Here are two options:

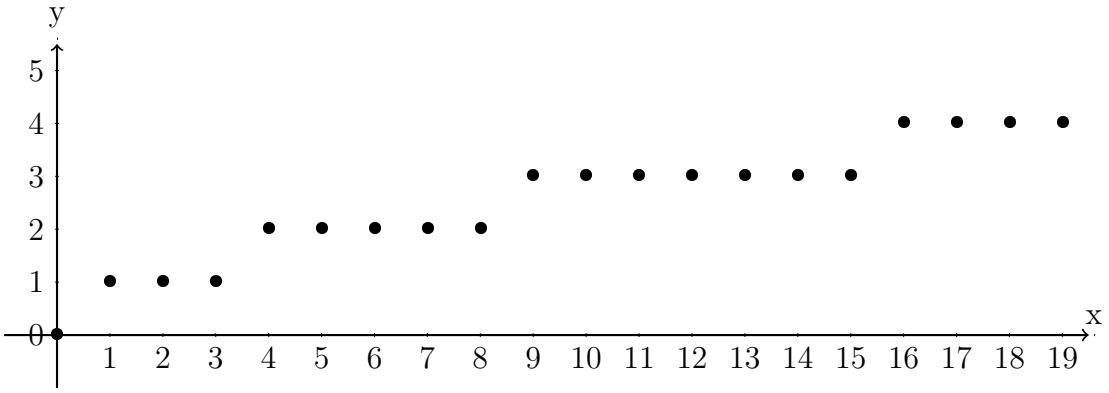
A. Take the real number square root and truncate to a natural number, e.g.

   $\sqrt{2} = 1.41421356237309504880168872420969807856967187537694 80$

   So root(2)=1. This would be hard, and might not work! Why not?

B. Define it intuitively. Consider what we mean.

   First we could draw the graph:



   What is the key property?

   Given x, we want the unique y such that $y^2 \leq x < (y+1)^2$.

   In logic $\forall x : \mathbb{N}.\exists y : \mathbb{N}.y^2 \leq x < (y+1)^2$.

   As a type, $x : \mathbb{N} \to y : \mathbb{N} \times (y^2 \leq x < (y+1)^2)$.

C. Imperative solution.

$y := 0$; $\{y^2 \leq x\}$ **while** $(y+1)^2 \leq x$ **do**

$\{(y+1)^2 \leq x\}$

$y := y + 1$

$\{y^2 \leq x\}$

od

$\{x < (y+1)^2\}$ loop exit condition

$\{y^2 \leq x\}$ the loop invariant

Here is another proposition that can be read as a programming task. These are two formulations of the same task, one using the quotient type. Think about how to write the program for the first version.

1. $\forall x{:}\mathbb{Z}.\exists y{:}\mathbb{Z}.(x = z * y \vee x = z * y + 1 \vee x = z * y + 2)$ or equivalently
2. $\forall x{:}\mathbb{Z}//\mathrm{mod}(3).b.(x =_z 0 \vee x =_z 1 \vee x =_z 2)$