Numerical Specifications and Programs: CS5860 Lecture 13 - 14

October 23, 2014

1 Introduction

These notes summarize ideas discussed up to and including Lecture 14. This material is related to Chapter 6 of the textbook by Thompson. In particular the idea of extracting a program from a proof is examined. The ideas discussed here take us deeper into the issues behind the design of constructive type theories such as the CTT of Nuprl and CIC of Coq that we have started to explore.

One of the main themes we are examining is central to understanding modern formal methods. On one hand, we have examined how to define numbers using first-order logic axioms. We have noted that weak first-order theories such as Q allow many non-standard interpretations of the numbers. There is a deep theorem of classical logic, the Lowenheim-Skolem theorem, that tells us that first-order logic is not adequate to define the *standard model* of the natural numbers. We will not prove this result from logic, but we will continue to discuss it informally.

It is no longer the case that first-order logic is considered the standard language in which to write formal specifications, so this issue is less urgent than in the past. Modern approaches to specification use higher order logic, HOL, or type theory.

We have already explored the notion that constructive type theory is based on a computation system rather than on pure logic. This is a major change in approach, and it puts programming languages closer to the heart of formal methods as in Thompson's textbook. We see this clearly in the short piece by Martin-Löf, and we will elaborate this theme as we go.

1.1 Overview

We already know that the very simple theory of numbers, Q, is too weak to characterize the natural numbers. It is also too weak to specify common numerical problems and prove that the specifications can be realized. We noted in Lecture 11 that we need some form of induction that is not present in Q. We continue this line of investigation here. We explore further the connection between *induction* and *primitive recursion*. We will show that the induction rule has intuitive computational content and is an instance of primitive recursion with functions as inputs. Thompson does a good job with this topic.

We will continue to explore Martin-Löf's approach to defining the natural numbers. As we mentioned previously, this will lead us to constructive type theory in a very natural way. In that setting, we will see the richest computational account of numbers available.

2 Heyting Arithmetic

The typical constructive axiomatization of a theory of natural numbers includes the numerical constants along with the successor, addition and multiplication functions axiomatized as in Q. In addition, the axiom scheme of induction is included for any first-order numerical predicate P of type $P : \mathbb{N} \to Prop$. The induction rule is typically presented as follows.

Induction Principle: Given any predicate $P : \mathbb{N} \to Prop$, we can construct evidence for the following relation,

$$P(0) \Rightarrow (\forall n : \mathbb{N}.(P(n) \Rightarrow P(S(n)))) \Rightarrow \forall n : \mathbb{N}.P(n).$$

The intuitive reason what we know $\forall x : \mathbb{N}.P(n)$ is that given any specific n, we can find evidence for $P(n)$ by starting with the evidence for $P(0)$, call it p_0, and applying the function f that takes any number i and evidence p_i for $P(i)$ and computes evidence $f(i)(p_i)$ to produce evidence for $P(S(i))$. We can do this repeatedly until we reach the number n and evidence for $P(n)$. For example, if n is $S(S(S(0)))$, then we build the following *evidence chain*: $f(0)(p_0)$ is evidence for $P(S(0))$, and $f(S(0))(f(0)(p_0))$ is evidence for $P(S(S(0)))$, and $f(S(S(0)))(f(S(0))(f(0)(p_0)))$ is evidence for $P(S(S(S(0))))$.

The approach we are developing to reasoning about the natural numbers is explained well in the textbook, *Type Theory and Functional Programming*, especially in section 4.8 starting on page 100. Thompson writes the key

induction rule on page 101 using basically the idea sketched just above. He stressed the connection to primitive recursion by introducing a primitive recursive operator with a function input, $prim\ n\ c\ f$ where $prim\ 0\ c\ f$ reduces to c and $prim\ S(n)\ c\ f$ reduces to $f\ n\ (prim\ n\ c\ f)$. The type of f is $n : \mathbb{N} \to P(n) \to P(S(n))$. This means that the *prim* operation takes a function as input. Its type is

$$n : \mathbb{N} \to c : P(0) \to (x : \mathbb{N} \to P(x) \to P(S(x))) \to P(n).$$

This typing is similar to that used in Nuprl for the induction operator. In the Nuprl case, induction is defined for the integers, $\mathbb{Z}$. So we can do induction going both *up* and *down*. The upward case can be seen as an example of the realizer given just above and the realizer in Thompson.

The Nuprl realizer for induction: In the base case, the Nuprl induction term $ind(0; t_b; x, y.t_{up})$ reduces to t_b. This is the *base case* for induction. This expression will have type $P(0)$ for the numerical predicate P.

In the induction case, $ind(s(n); t_b; x, y.t_{up})$ reduces to $t_{up}(s(n))(ind(n; x, y.t_{up}))$.

The type of the induction operator *ind* is

$$P(0) \to (u : \{i : \mathbb{N} | 0 < i\} \to P(u - 1) \to P(u)) \to (n : \mathbb{N} \to P(n)).$$

On page 102, in discussing how induction is expressed in type theory, Thompson says:"This rule is one of the highlights of type theory." This judgment reflects the insights of Martin-Löf in designing intuitionistic type theory using insights about induction developed by Skolem, Robinson, Gödel, Goodstein [1], Tait and others who explained induction in computational terms and expressed the rules using type theory.

2.1 Primitive Recursive Functions of Higher Type

In Lectures 11 and 12 we examined primitive recursive functions whose only inputs are numbers. To understand induction, it has been useful to generalize our definition to include inputs that are themselves computable functions. This investigation was undertaken by Gödel to provide a computational interpretation of number theory. His system is called T in some articles and is referred to as his Dialectica interpretation based on the name of the journal in which his article was published. It is nicely summarized in the book by Stenlund, *Combinators, λ–Terms, and Proof Theory*, [2]. Tait gave an elegant method for proving that these functions terminate [3]. This work by Tait was another major influence on the development of modern type theory.

It is an interesting exercise to extend Martin-Löf's account of primitive recursion to this larger class of functions. In his notes posted for the course, one of the key insights is expressed after he has given a precise syntax for the language of primitive recursive functions, and he says:

"With this, the formulation of the syntax of the language of primitive recursive functions is complete. ... It remains for us to explain the *meanings* of the statements that can be derived by means of the formal rules (or, what amounts to the same, how they are understood) because understanding a language, even a formal one, is not merely to understand its rules as rules of symbol manipulation. Believing that is the mistake of formalism."

The first challenge for understanding modern type theory is to understand these higher-order recursive functions. We see here that such an understanding is important even for arithmetic. An interesting course project would be to give a Martin-Löf style account of natural numbers that uses this notion of numerical function of higher type to explain induction.

The challenge of induction We have now established a strong connection between induction and primitive recursion. On the other hand, we have remarked that there is another way to grasp induction that does not carry the computational meaning, yet it can be used to justify properties of primitive recursion. Here is the example discussed in Lecture 12. It would be interesting to explore this idea further as a contrast to the computational interpretation.

Addition is a total function:
$\forall x, y : \mathbb{N}.\exists z : \mathbb{N}.(add(x, y) = z)$.

The proof is by induction on y. In the base case, $y = 0$ we see that $z = x$. If we assume $\exists z : \mathbb{N}.(add(x, y) = z)$, then as we saw in lecture it is possible to prove by induction $\exists z : \mathbb{N}.(add(x, S(y)) = z)$.

In a way this defeats the reason Skolem used primitive recursive functions, on the other hand, it shows that induction provides an explanation for termination of the *add* function. In this use of induction, we don't need the computational content. Essentially we are using induction as a way of type checking the primitive recursive function. When we come to studying constructive type theory, we will see the Nuprl uses induction as a way of proving that formulas are well formed and that functions are well typed. Proving that $\lambda(x, y.add(x, y))$ has type $x : \mathbb{N} \to y : \mathbb{N} \to \mathbb{N}$ is done by induction on y.

Exercises. In Lecture 12 we discussed some of the questions that will appear on Problem Set 4. One of them is to finish this proof and discuss

its contribution to believing that the addition function is total. Note that the argument applies equally well to all of the primitive recursive functions discussed above. The problem set investigates the form of the induction realizer for this simple proof.

Another topic we mentioned in lecture is that the definition of the exponential function is somewhat more complex than for addition and multiplication. Can you see this extra complexity. The Princeton mathematician Edward Nelson believes that the exponential function should not be considered computable. Can you think of a reason for his opinion?

We will explore this idea indirectly in Problem Set 4.

3 Other Induction Principles

Another form of induction, called *complete induction* or *course-of-values induction*, is more useful than standard induction and can be derived from it. Here is one of the ways it is expressed.

Complete Induction-1 (Course-of-Values Induction):

$$\forall x.(\forall y.(y < x \Rightarrow A(y)) \Rightarrow A(x)) \Rightarrow \forall x.A(x).$$

Another form of this induction is a bit more intuitive, yet equivalent.

Complete Induction-Explicit-Base:

$$A(0)\&\forall x.(\forall y.(y \leq x \Rightarrow A(y)) \Rightarrow A(x+1)) \Rightarrow \forall x.A(x).$$

For the first form of complete induction, the base case, $A(0)$ follows from the fact that $\forall y.(y < 0 \Rightarrow A(y)) \Rightarrow A(0)$ is required to hold (instantiating x with 0), and since it is the case that $\forall y : \mathbb{N}.(y < 0 \Rightarrow A(y))$, since there are no such y, it must be that $A(0)$ holds. So we have the base case "by default" so to speak.[1]

To prove Complete Induction-Explicit-Base, we assume

$$A(0)\&\forall x.(\forall y.(y \leq x \Rightarrow A(y)) \Rightarrow A(x+1))$$

and then prove $\forall x.\forall y.(y \leq x \Rightarrow A(y))$ by *simple induction* on x.

From this, $\forall x : \mathbb{N}.A(x)$ follows by taking y to be x. That is a simple sketch. It breaks the result into two small lemmas. The first one captures the critical insight. We call it the Main Lemma.

[1]The fact that we use *ex falso quodlibet* in the base case is a bit inelegant, so the form with Base might be preferable.

Main Lemma:

$\vdash A(0)\&\forall x.(\forall y.(y \leq x \Rightarrow A(y)) \Rightarrow A(x+1)) \Rightarrow (\forall x.\forall y.(y \leq x \Rightarrow A(y)))$.

We immediately obtain this sequent:

$hyp : (A(0)\&\forall x.(\forall y.(y \leq x \Rightarrow A(y)) \Rightarrow A(x+1))) \vdash \forall x.\forall y.(y \leq x \Rightarrow A(y))$

We can decompose the hypothesis into

$a_0 : A(0), all : \forall x.(\forall y.(y \leq x \Rightarrow A(y)) \Rightarrow A(x+1))), x : \mathbb{N}$

$$\vdash \forall y.(y \leq x \Rightarrow A(y)) \ by \ \lambda(x.ind(x;\ ...; u, v.\ ...))$$

$$\vdash \forall y.(y \leq 0 \Rightarrow A(y)) \ by \ \lambda(y.\lambda(le.\ ...))$$

$y : \mathbb{N}, le : y \leq 0 \vdash A(y) \ by \ arith \ y = 0$
and the subproof
$y : \mathbb{N}, le : y \leq 0 \vdash A(0) \ by \ a_0$

Next, for the induction case, we assume the result for x and prove it for $x+1$.

$a_0 : A(0), all : \forall x.(\forall y.(y \leq x...), x, u : \mathbb{N}, v : \forall y.(y \leq x \Rightarrow A(y))$
$\vdash \forall y.(y \leq x+1 \Rightarrow A(y)) \ by \ \lambda(y....)$

$$y : \mathbb{N}, ll : y \leq x+1 \vdash A(y) \ by \ all(x, y).$$

The small lemma is simply this:
Small Lemma: $\vdash (\forall x.\forall y.(y \leq x \Rightarrow A(y))) \Rightarrow \forall x.A(x)$.

This is very simple to prove since we have $x \leq x$.

The realizer is easier to understand if we use the subtyping notation where $\mathbb{N}_x$ is the type $\{n : \mathbb{N} | n \leq x\}$. We can then give the type of complete induction as follows.

$$A(0)\&\forall x : \mathbb{N}.((\forall y : \mathbb{N}_x.(A(y)) \Rightarrow A(x+1)) \Rightarrow \forall x : \mathbb{N}.A(x).$$

If we let $all : (\forall x : \mathbb{N}.((\forall y : \mathbb{N}_x.A(y) \Rightarrow A(x+1))))$, then we can see that $all(x) \in (\forall y : \mathbb{N}_x.(A(y)) \Rightarrow A(x+1)$. Thus if $g_x \in y : \mathbb{N}_x \rightarrow A(y)$, then $all(x)(g_x) \in A(x+1)$. Thus $all(0)(g_0) \in A(1)$ and $all(1)(g_1) \in A(2)$ and so forth.

We can prove complete induction from simple induction and conversely. On the other hand, the two principles have distinct realizers. We will see that complete induction can be used to build forms of *fast induction*. The main idea of complete induction is that in proving $A(n+1)$, we assume that we have evidence for all of $A(0), A(1), ..., A(n)$, so we can use whatever of these values we need to provide evidence for $A(n+1)$. We know by standard (Peano) induction that using only $A(n)$ is sufficient, but we might have a simpler argument for certain instances of A if we use other values as allowed by complete induction. This induction method is *complete* in the sense that we have the complete initial segment of values at our disposal for building evidence for $A(n+1)$.

4 Related Theorems

Sometimes the method of inductive proof is captured in a negative way, using the method of infinite descent.

Theorem of Infinite Descent:

$$\forall x : \mathbb{N}.(A(x) \Rightarrow \exists y : \mathbb{N}.(y < x) \& A(y)) \Rightarrow \sim A(x).$$

$$\forall x : \mathbb{N}.(A(x) \Rightarrow \sim\sim \exists y : \mathbb{N}.(y < x) \& A(y)) \Rightarrow \sim A(x).$$

Here is another basic principle used to prove properties of the natural numbers. The first version is not a valid computational procedure, but it commonly found in mathematics texts.

Least Number Principle on Decidable Predicates:

$$\forall x.(A(x) \vee \sim A(x)) \Rightarrow (\exists x.A(x)) \Rightarrow \exists y.(A(y) \& \forall z.z < y \Rightarrow \sim A(z)).$$

We can imagine a simpler principle if we work in a sub-logic in which we do not keep track of all the evidence. This kind of sub-logic is called classical logic, and we will examine it more later in the course.

{**Least Number Principle**:}

$$\exists x : \mathbb{N}.A(x) \Rightarrow \exists y : \mathbb{N}.(A(y) \& \forall z : \mathbb{N}.z < y \Rightarrow \sim A(z)).$$

References

[1] R. L. Goodstein. *Recursive Number Theory.* North-Holland, Amsterdam, 1957.

[2] S. Stenlund. *Combinators, λ–Terms, and Proof Theory.* D. Reidel, Dordrechte, 1972.

[3] W. W. Tait. Intensional interpretation of functionals of finite type. *The Journal of Symbolic Logic*, 32(2):189–212, 1967.