Lecture 13

Midterm reminder Friday March 6 – 2 weeks from today

1 Overview

Lecture 10 by Abhishek Andand was on the Coq system and the *Coq programming language*, CoqPL. Then Dr. Rahli devoted Lectures 11 and 12 (interrupted by President's Day) to the topic of evaluating lambda terms, and on Monday February 23 he will finish the topic in Lecture 14. His lecture will prepare you to study the 2003 BRICS article by Danvy and his colleagues on implementing Abstract State Machines for the lambda calculus if you wish to investigate this topic more deeply. We will have taken you from the theory of the lambda calculus and combinators to their implementation in modern functional programming languages such as OCaml and SML (Standard ML). That series of lectures essentially finishes our account of the *untyped Lambda Calculus* and Combinators by "bringing it down to earth" and allowing those who like to learn by implementing a chance to write your own pure lambda calculus state machine.

I want to use today's lecture to tie together the topics of Lectures 10, 11, 12, and 14 and put them in context so that you can see the connections to the rest of the course and its main themes. Doing this will also help prepare you for Lecture 14. I also expect that doing this will help you with Problem Set 2 and the questions about primitive recursion. The structure of this lecture is partly review and context.

We will recall the four models of computing that we have studied: the Lambda Calculus, Combinators, Primitive Recursion, and Kleene's Partial Recursive Functions (using Primitive Recursion and the μ-operator (least number operator)). We will classify these models into the *Turing complete* models and the *subrecursive* models. That classification is important for this version of the course since most offerings of CS6110 do not devote much time to subrecursive languages. The existence of CoqPL has changed that. Moreover, I always cover the topic since subrecursive languages provide a very nice connection to Theory A, and I am keen to foster connections between these two basic branches of computing theory. Indeed, my dream for years has been to provide a type theory foundation for computer science that will unify these fundamental areas of CS. I am far from alone in this hope. Professor Kozen is a living example of this unity.

In addition, this lecture will be our first effort to contrast *methods of defining programming languages*. The topic is item 4 in the list below, and in item 5 we make the contrast to the semantic methods for the lambda calculus. It is already clear that the methods used for

the Lambda Calculus are quite different than those for the primitive recursive and partial recursive functions. This difference is fundamental. The Lambda Calculus semantics involves the use of a *metalanguage* which is very close to the language being defined. The primitive and partial recursive function definitions do not use this methodology.

The metalanguage approach is very manifest in the programming language Lisp and efforts to provide a firm formal semantics for that language. It gives Lisp the power tointernally define new languages. This is done using the QUOTE operator to talk about Lisp syntax in the language itself. The Lisp idea is similar to mechanisms used in natural languages to talk about themselves.

Quoting and the use of metalanguage are closely related. I think that there is more research to be done on understanding the importance of the differences and the issues with the notion of a formal metalanguage. These are significant and deep ideas in my view, and they are not fully understood. This lack of a complete understanding is due to the fact that we can find ways to avoid this approach or minimize it. The method used for *partial recursive functions à la* Kleene is one.

To illuminate this important and subtle point, I want to also bring into the story in more detail the efforts of Herbrand and Gödel to define the notion of *general recursive functions* or Herbrand-Gödel recursive functions. I make brief remarks about this now and then elaborate in Lecture 16 next Wednesday, February 25. Next week we will examine small fragments of the basic articles on this topic by Gödel and Kleene. Their ideas are easy to state and remarkably insightful. They also connect to the early history of the effort to understand the computable functions. In due course we might look at the method that the logicians use to reduce reflection and metalanguage to the idea used by Gödel, it is called *arithmetization* or *Gödel numbering*.

Already the lectures of Dr. Rahli have revealed the critical nature of the metalanguage. We see how that concept is used at a high level and then carried all the way down to the code in the BRICS implementations. The fact that the metalanguage resembles the object language, the Lambda Calculus, is already on the face of it, fascinating to those who like to think about the basic mechanisms of human language.

2 Lecture Topics

1. The PL models so far:
   - λ-calculus (Universal, i.e Turing complete)[1]
   - Combinators (Universal)
   - Primitive recursion and CoqPL (Subrecursive languages)
   - Partial recursive functions (Turing complete)
     - μ-operator and Primitive Recursion
     - Herbrand/Gödel recursive [2]

[1] Turing was careful to allow *oracles*, that is part of some models explicitly. What about for the λ-calculus? Add a function term f with oracular reductions. Have non-deterministic combinator been studied? Yes! α

2. Functional PL mechanisms so far:

   Howe's equality
   Environments, closures
   Continuations (tail recursion)
   (Path to compilation)
   Yet to come – defunctionalization, Abstract State Machines
   "Compiler Theory" – the "logic of compilers"

3. Relating models.

   All can be *compiled to Turing machines* or other universal machine models: RAM, RASP, G_3 machine, etc. If they can simulate a Turing machine, they are *universal*. We should also require *oracles*. This has become key in Nuprl's event logic. We also call these models *Turing complete*.

4. Methods of definition.

   Primitive and Partial Recursive defined using inductively defined classes of functions and partial functions vs. λ-terms. This will be a separate section.

   Recursive functions in mathematics, refining this idea led to general recursive functions or Herbrand/Gödel recursive functions, partly captured in CoqPL. Primitive recursion is natural, but see PS2 on the Fibonacci function.

   Primitive recursive functions are *tail recursive*.

5. The lambda calculus theory requires a *meta language* as we see in Dr. Rahli's lectures. This seems circular to use a lambda calculus implementation to define the lambda calculus evaluator. Is this whole method sound? Or are we just "kicking out the chair" after we have created a compiler for our language by standing on it? What is the logic behind compilers and interpreters?

6. Steps toward the *BRICS evaluators* (in SML)

3 Methods of defining computable functions.

The first glimpse of the notion of a universal class of computable functions came from the study of recursive functions in number theory. It was common since at least Gauss to use recursive functions and induction to state and prove results in number theory. Indeed there are efforts to state the results of number theory very formally using recursive functions, many of which are primitive recursive, all of which are Herbrand-Gödel recursive.

Recursive Functions by Rozsa Peter, 1967 from her work starting in 1932.

$+ \beta$.

[2]This example is very special. It appears to provide all the total computable functions without introducing partial functions. Can this be? Or must the language be subrecursive in some way?

Herbrand 1932
Herbrand Gödel 1934
} General recursive functions

Kleene 1936, wrote key papers relating general recursive functions to his partial recursive functions defined in lecture. We will look at excerpts of one of these along with his notes with Rosser on Gödel's lectures.

The λ-calculus ideas came from Church in 1932 as well, revisited in 1940.

Turing 1937 – Turing machines [Gödel said: this definition of computability is *absolute*, perhaps the only absolute idea in logic. It is not entirely clear how to be precise about this notion of absoluteness. It seems to suggest something foundational that transcends any particular formalism. This informal insight might be key to why a modern constructive type theory of partial computable functions, might be an adequate foundation for both mathematics and computer science. This is one of my favorite ideas and themes.]

1937 – PhD compilation of λ-calculus to Turing machines.

Markov algorithms 1947 – other late models of computersl RAMS, RASP, etc.

The recursive function style is mathematical or *denotational*, based on classes of functions and methods of recursive definition. The use of recursion along with very simple operations such as composition and projection seems to be the core of computation. This idea gradually arose a logicians studied the forms of recursive definition.

Primitive (singly recursive)

Double recursive, course-of-values, simultaneous

n-recursive $\quad h(p+1, n+1, a) = h(p, h(p+1, n, a), a)$ $\quad$ See Ackermann example of a doubly recursive function.

Higher-type recursion on *functionals* will be discarded later. They are used in Gödel's system T – 1958

Recursive Number theory – Goodstein 1957 shows how to formulate results in number theory, such as the fundamental theorem of arithmetic, using recursive functions. He later did elements of analysis this way (Recursive Analysis).

What is the essence of this method?

Closing a class of basic functions under operations. The base functions are all obviously computable, and recursion seems to be sufficient to obtain all of the computable functions used in mathematics. No "syntax" or "metalanguage" is needed, just mathematical operations cast as functions and the notion of an inductive class of functions using *computable functionals*, i.e. functions taking functions as inputs and outputs.

We can experience these ideas by studying the class of primitive recursive functions and looking at how they can be extended by defining the Ackermann function and other "general forms" of recursive definition. Herbrand suggested allowing any recursive definition as long as it defines a function in some constructive sense.

4 Steps toward the BRICS evaluators

A key step in Dr. Rahli's explanation of lambda calculus evaluators is transforming the natural recursive evaluator to a tail recursive version. To illustrate this in a simple setting, let's notice that all primitive recursive functions are tail recursive. We show where the terminology comes from by implementing these recursions using while loops. This implementation shows what a stack frame is and how they are collapsed in implementing primitive recursion with a while loop. Consider the following simple example.

$$\begin{cases} f(0,y) = a(y) \\ f(S(n),y) = h(n,y,f(n,y)) \end{cases}$$

Consider this computation:

$$\left.\begin{array}{l} f(3,5) = h(2,5,f(2,5)) \\ f(2,5) = h(1,5,f(1,5)) \\ f(1,5) = h(0,5,f(0,5)) \\ f(0,5) = a(5) \end{array}\right\} \text{ These are } \textit{stack frames}$$

We can see how destructive updates can collapse the *stack frames* – using "mutable variables", i.e. imperative programs. We study their semantics later in the course.

$i := 0;\ f := a(5)$
<u>while</u> $(i < n)$ <u>do</u>
    $f := h(i,5,f)$
    $i := i+1$
<u>od</u> $\{f(i,5) = h(i-1,5,f(i-1,5))\}$
$\{i = n\}$
$\{f(n,5) = h(n-1,y,f(n-1,y))\}$

The while loop does tail recursion.

Exercise: Write the while loop as a recursive function. (To be assigned in PS3.)

Steps toward understanding the BRICS (Basic Research in CS, Aarhus, Denmark) paper on the course web page: *A Functional Correspondence Between Evaluators and Abstract Machines*. Ager, Bienacki, Danvy, Midtgaard, 2003.

They write a sequence of evaluators in SML. Eval0 produces Krivine's Abstract State machine. It uses λ-terms coded with *deBruijn* λ-terms. The bound variables are given by their off-set numbers from the binding λ. For example:

$\lambda x.\lambda y.x$ is $\lambda\lambda 2$
The S-combinator $\lambda x\lambda y\lambda z.xz(yz)$ is $\lambda\lambda\lambda 31(21)$
$\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx)$ is $\lambda(\lambda 1)(\lambda 1)(\lambda 21)$

You only need to know that bound variables are numbers. We won't use this notation. Substitution is nasty, need to renumber.