Tex Notes on "Lecture Notes on The Lambda Calculus"

June 19, 2022

Contents

1 Introduction

Notes based on the "Lecture notes on the Lambda Calculus" by Peter Selinger.

We begin by a distinction between two paradigms of thinking about functions; Intensional vs Extensional.

Extensional The extensional view is that of mathematics. This is the view of a function as a set $f \subseteq X \times Y$. This allows for more functions than the intensional paradigm (because not all of these will be given by a rule)

Intensional This is the paradigm more appropriate to computer science and considers functions as rules for computing an output from a given input. This will be the paradigm that allows you to think about how an output is calculated, in how much time and using how many resources.

The Lambda calculus is essentially a language that can be used to express functions as rules. Although this is an extra-system interpretation that is not essential to the definition of the lambda calculus.

If we consider typed lambda calc terms we see that they are clearly a strict subset (is the collection of all functions a set, cant be) of the mathematical notion of functions, does it make sense to try to compare the untyped lambda calc to the collection of mathematical functions. I suspect not.
I suspect that they are not sets properly so I would need some other machinery to handle this question.
What about if I restrict to a suitable smaller class of functions (and typed terms), how would I interpret the function

$$f : \mathbb{R} \to \mathbb{R}, x \to x^2$$

say as a typed lambda term, would I make the type of the term $\lambda x.x^2$ $\mathbb{R}$?
Come back and see if any of this is coherent.

1.1 Typed VS Untyped

The untyped lambda calculus how no notion of domain or codomain and so allows you to generate mathematically nonsensical functions, the typed lambda calculus always fully specifies both and so brings you very close to the situation in mathematics.

There is an intermediate system known as polymorphically typed. This allows you to specify for example that a term has type $X \to X$ without specifying an X.

1.1.1 Computability

Turing, Church and Godel independently defined different but later proven equivalent notions of what it means for something to be computable.

What's the history here. Sounds interesting.
Why did they chose their respective formalisms and is it as interesting as it sounds that they all turned out to be the same. It seems like a mighty coincidence.

The lambda calculus is Turing complete.

This means it "realises" a turing machine, I need to revise what this means exactly

1.2 Constructivism

Constructivists believe that to prove the existence of a mathematical object one must give an explicit algorithm for creating said object. Classical logicians to contrast believe it is sufficient to derive a contradiction from a statements negation to show that it holds.

Because the lambda calculus gives a way to describe algorithms (functions as rules) one should not be surprised to find out that it can be used to describe intuitionistic proofs. This idea is made concrete in the Curry-Howard isomorphism.

2 The Untyped Lambda Calculus

The lambda calculus is a formal language. That is a collection of words formed from letters of a given alphabet in accordance with certain rules.

Something that always disturbs me when people start defining formal languages is that they seem to rely heavily on mathamatical concepts. But it seems to me that people have attempted to make mathematics rigorous only through making it a formal language, I have long been bothered by the apparent circularity yet hope that im missing something subtle that will make it at least not such an egregious circularity.
For instance what is the status of sets, cardinality, quantification and functions.
We need countable sets of variables, we quantify over terms in the calculus (or in formal languages more generally, and given it is countably infinite it must be genuine quantification and not merely shorthand for enumerating all FINITE combinations), we have our grammas specified by functions.
I think in Will and Billys foundations they talked about this a little bit, return to it and meditate on their comments.

We can express formal languages that are generated by a context free grammar using the Backus-Naur form

$$< symbols >::== expression_1|...|expression_n$$

Where the symbols on the left can be replaced by one of the expressions on the right to get a well formed formula.

Am I right about this in fact being only for context free grammars. Is there a restriction on the cardinality of the set of exchange rules in a context free grammar (in particular is it finite?)

In this notation we can define the lambda calculus

Definition 1

Given an infinite set V of variables (the elemnts of which we will typically denote $x, y, z...$) the set of lambda terms is given by

$$M, N ::== x|(MN)|(\lambda x.M)$$

Definition 2

Equivilently; given an infinite set V of variables, and A an alphabet consisting of the elements of V and special symbols ")","(","λ",".". Let A^* be the set of strings (finite setquences) over the alphabet A. Then the set of lambda terms will be $\Lambda \subseteq A^*$ the smallest subset such that

- $x \in V \implies x \in \Lambda$
- $M, N \in \Lambda \implies (MN) \in \Lambda$
- $x \in V, M \in \Lambda \implies (\lambda x.M) \in \Lambda$

Later he will talk about models of the lambda calc and that set theory is a sound and complete model I think, however from this definition it seems like its not just a model, these definitions construct the lambda calculus as a set theoretic object, the triple (V, A, Λ).

This definition ensures that each term (through the use of brackets) can be uniquely decomposed into into subterms. Moreover each $M \in \Lambda$ is of the form $x, (AB), \lambda x.A$ called variable, applications and lambda abstractions respectively.

We have the following further conventions to make the notation less cumbersome:

- We write (MN) instead of $M(N)$ for the application of a "function" to another term.
- We omit the outermost parenthses
- Applications associate to the left e.g. $fxyz = ((fx)y)z$
- The body of the lambda abstraction (the section after the dot) extends as far right as possible
- We can contract lambda abstractions e.g. $\lambda x.\lambda y.\lambda z.M = \lambda xyz.M$

Is there a meaningful way to show that these two systems are actually the same.
What does it mean for two formal languages to be the same actually.
I mean its clear that lambda with these rules is not litterally the same as the "official" definition (there a terms in one that are not in the other) so there must be a sense in which we can map between them preserving all the structure (which is what)?

2.1 α-Equivalence

We now begin simplyfing the lambda calculus by looking at its equivilence classes under different relations. The first (α equivilence) captures that two terms are essentially the same if all we have done is change the characters that we are using.

What we mean by the same here is the same to a human interpreter, because within the lambda calc they are very much distinct objects and a priori we have no interpretation on the terms to allow us to say that any two strings in the language are "the same".

To make this precise takes quite some work however.

2.1.1 Bound Variables

We say that an occurence of a variable x inside a lambda abstraction $\lambda x.N$ is bound. λx is the binder.The term N is the scope of the binder. A variable that is not bound is free.

Definition 3

The set of free variables of a term M is denoted $FV(M)$:

- $FV(x) = \{x\}$
- $FV(MN) = FV(M) \cup FV(N)$
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$

2.1.2 Replacement

Definition 4

We define and denote the renaming of a variable x with the variable y as (where $z \neq x$):

- $x\{y/x\} = y$
- $z\{y/x\} = z$
- $(MN)\{y/x\} = (M\{y/x\})(N\{y/x\})$
- $(\lambda x.M)\{y/x\} = \lambda y.(M\{y/x\})$
- $(\lambda z.M)\{y/x\} = \lambda z.(M\{y/x\})$

This replaces all occurence (free or bound) of x with y.

2.1.3 Equivilence

α-equivilence is then the smallest congruence relation (reflexive, symmetric, transitive, and two additional rules set out below) on lambda terms such that for all terms M, and all variables y that do not occur in M

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\})$$

α-Equiv is the smallest relation satisfying the following rules

$$(\textit{refl}) \quad \frac{}{M = M} \qquad (\textit{cong}) \quad \frac{M = M' \qquad N = N'}{MN = M'N'}$$

$$(\textit{symm}) \quad \frac{M = N}{N = M} \qquad (\xi) \quad \frac{M = M'}{\lambda x.M = \lambda x.M'}$$

$$(\textit{trans}) \quad \frac{M = N \qquad N = P}{M = P} \qquad (\alpha) \quad \frac{y \notin M}{\lambda x.M = \lambda y.(M\{y/x\})}$$

In practice we can and will assume that bound variables have been renamed to be distinct (this is without loss of generality).

Is this just exactly what working in the alpha equiv lambda calc give us

2.2 β Equivilence

Analogous to renaming we want to be able to replace a variable by a more complicated lambda term.

We need to be more careful because we only want to substitute for free variables, and we dont want to accidentaly bind any free variables.

We only substitute for free variabels because in a sense the binder (lambda abstraction) makes the label unimportant.

To make sure we dont capture variables we use an unused varibale (called fresh), this is why we require an infinite set of variables in our definition, so there are always fresh variables available to us.

Definition 5

The capture avoiding substitution of N for free occurrences of x in M; denoted $M[N/x]$ is given recursively as follows:

-

We have not given a well definied map becuase we dont specify how to choose the fresh variable. We could order the set of variables and give a rule that uniquly gives a new variable, however we dont need to if we are working modulo α equivilence (its actully well definied modulo α equivilence).

Is there a bijection between lambda calc and lambda calc mod $=_\alpha$ as sets?
If so is there any difference detectable between these structures.

We will now talk only up to α equivlience and often will write this as $=$.

2.3 β Reduction

3 Programming in the Untyped Lambda Calculus

4 Church - Rosser Theorem

5 Combinatory Algebras

6 Simply Typed Lambda Calculus, Propositional Logic and the Curry-Howard Isomorphism

7 Weak and Strong Normalization

8 Polymorphism

9 Type Inference

10 Denotational Semantics

11 The Language PCF

12 Complete Partial Orders

13 Denotational Semantics of PCF

Index