Computational Calculus Primer Part 1: Lambda calculus & BNF

0.1 Overview

Greg Meredith has been teaching a weekly course called *Introduction to the Design of Computational Calculi (DoCC)* in an effort to explain the foundations of mathematical models of grammar, how computational calculi are used in the RChain project, and how to design computational calculi for specific applications. As this is a highly technical topic, it may seem like the ideas are exotic or esoteric and difficult to wrap your head around. However, I am here to help you understand these ideas. I will make them as down-to-earth and comprehensible as possible while maintaining rigor and keeping in mind the higher purpose.

The first few posts here will serve as background material required for following along with the course. We first need to cover some of the basics of computational calculi, like the syntax and semantics of λ-calculus, π-calculus, and the model RChain is built on, ρ-calculus. After covering these basics, there will be weekly posts explaining the DoCC material in more depth.

First, we all need to get comfortable with BNF and λ-calculus. So we will start our discussion with grammar and operational semantics.

0.2 Baukus-Naur form (BNF)

The grammar we discuss will always be presented in what is known as *Baukus-Naur form (BNF)* which is typical for programming languages. BNF is just a concise way of describing how the rules for the syntax work in a grammar, i.e. what "admissible" terms look like in the language. The BNF presentation of a grammar explicitly displays the syntactic rules. To keep the discussion concrete, let's look at the example of λ-calculus.

1 Lambda calculus

1.1 Grammar

The λ-calculus is a universal model of computation, discovered by mathematician Alonzo Church in the 1930's. The BNF presentation of a grammar for λ-calculus:

$$M, N ::= \ x \mid \lambda x.M \mid (MN) \tag{1.1}$$

You don't actually need to know what these things mean yet, you just need to know that there are *three* basic forms for terms which correspond to the three different rules for constructing terms. This presentation gives rise to a *language*, the collection of λ-terms, i.e. terms of the form described

on the righthand side of the BNF presentation above. Our language consists of terms we call M and N of the form x or $\lambda x.M$ or (MN). Notice that these terms are built recursively so we actually have quite elaborate terms in our language. Keep in mind, we also get all admissible combinations of these forms in the language too.

It's a game. In this game, there are three rules describing how you may construct terms and a hat to draw names from. Each round, you have a λ-term M to work from and you are given the choices: (i) draw a name x from the hat, (ii) draw a name x and form the abstraction $\lambda x.M$, (iii) choose another λ-term N and form the application (MN). That's it, now you can build any λ-term by playing sufficiently many rounds of this game.

This begs the question of where we start. Well, we can always choose a variable x from our variable set X, so let's start there. Let's make our first λ-term x. Then from this term, we can form other λ-terms, notably, $\lambda x.x$ and $(x\ x)$. Now we are off to the races! What other λ-terms can be formed?

Returning to the BNF presentation of λ-calculus (1.1). One noteworthy thing is the appearance of x on the right-hand side. Where did that come from (there is only M and N on the left-hand side)? Well, it turns out that x is just a *name* or *variable* we choose from out of a set X. This set X is something we have control over and changing it changes the admissible expressions in our language. Therefore, we say that this grammar is *paramteric* in the set of variables X. This leads some people to write the BNF presentation of λ-calculus in a slightly odd looking format with the variable set X displayed explicitly

$$M[X] ::= \ X \mid \lambda X.M[X] \mid (M[X]\ M[X])$$

to emphasize the dependence of the λ-terms $M[X]$ on X. But I digress. The important thing here is that the λ-terms depend on the given variables.

Another noteworthy aspect of (1.1) is the appearance of Ms and Ns on both sides of the BNF presentation. Basically, what this means is that if we have two λ-terms M and N, we have a few options for creating new λ-terms, recursively:

(i) x (*mention*; pick a variable from the set X, admittedly this has little to do with M and N);

(ii) $\lambda x.M$ (*abstraction*; binds the variable x in M, i.e. declares x a variable in the function M);

(iii) (MN) (*application*; "applies" the function M to the argument N)

That's basically it. Pretty simple, huh? We always have the option of forming these admissible terms given some admissible term(s). Let's look at a few examples.

Examples:

1. $\lambda x.\lambda y.x$ is admissible (this is a really important example because it is the encoding of the Boolean function True in the λ-calculus)
   Proof: x is admissible $\implies$ $M = \lambda y.x$ admissible $\implies$ $\lambda x.M = \lambda x.\lambda y.x$ admissible

2. $(\lambda x.x\ y)$ is admissible (this is also an important example because $\lambda x.x$ acts as the identity function in λ-calculus)
   Proof: $\lambda x.x$ admissible $\implies$ $(\lambda x.x\ y)$ admissible (application of $\lambda x.x$ to y)

3. $x\ \lambda x.$ is not proper syntax $\implies$ not an admissible λ-term

This is all we need to know about constructing λ-terms. However, this doesn't tell us anything about how these terms interact with one another. For that we will need to discuss the operational semantics.

1.2 Operational semantics

The λ-calculus captures the notion of a function; abstractions picking variables in the function and applications evaluating the function at the variable. This thinking guides the interactions between these constructions, known as the *operational semantics*. There are two main relations in λ-calculus, α-conversion and β-reduction:

- *α-conversion*:
$$\lambda x.M\{x\} \to \lambda y.M\{y\}$$
This rule deems terms that only differ by the names of their *bound* variables *structurally equivalent*. By structurally equivalent, we mean that these terms are indistinguishable from the viewpoint of computation. By bound variable, we mean something like x in the expression $\lambda x.M\{x\}$. The variable x appears in the term M (that's why it's being written like $M\{x\}$) and λ binds x in this expression, i.e. λ declares x a variable in M. Another way to think about this is that a function f can have x as a variable, we write $f(x)$, or it can have y as a variable, we write $f(y)$, and regardless of what we call the variable, this function denotes/does the same thing. This is the essence of α-conversion. This is an equivalence of terms of the form $\lambda x.M$ and $\lambda y.M\{y/x\}$.

- *β-reduction*:
$$(\lambda x.MN) \to M\{N/x\}$$
This rule reduces an application of the form $(\lambda x.MN)$, where the first term is a λ-abstraction, $\lambda x.M$, to a substitution of the term N for the variable x in the term M. This is simply function evaluation. We are declaring x a variable in the function M and evaluating it at N, i.e. substituting N for x. This can also be thought of as function composition. If we apply the function $f(x)$ to the function $g(y)$, we get the composition $f(g(y))$. This is exactly the same as substituting $x = g(y)$.

Examples:

1. $\lambda x.(x\ z) \to \lambda y.(y\ z)$
Let $M\{x\} = (x\ z)$ and $M\{y\} = (y\ z)$ so we have $\lambda x.(x\ z) = \lambda x.M\{x\} \to \lambda y.M\{y\} = \lambda y.(y\ z)$ by direct α-conversion. Even though there is abstraction and application in the term $\lambda x.(x\ y)$, it does not β-reduce. Why not?

2. $(\lambda x.(x\ x)\ y) \to (y\ y)$
This is a β-reduction because we are applying the abstraction $\lambda x.(x\ x)$ to the term y. Let $M\{x\} = (x\ x)$. Then $(\lambda x.(x\ x)\ y) = (\lambda x.M\ y) \to M\{y/x\} = (x\ x)\{y/x\} = (y\ y)$ by direct β-reduction.

3. $(\lambda x.(\lambda y.(x\ y)\ u)\ v) \to (v\ u)$
There are two rounds of β-reduction here. Let $M\{x\} = (\lambda y.(x\ y)\ u)$. Then $(\lambda x.(\lambda y.(x\ y)\ u)\ v) = (\lambda x.M\ v) \to M\{v/x\} = (\lambda y.(x\ y)\ u)\{v/x\} = (\lambda y.(v\ y)\ u)$. Now we use a direct β-reduction on the remaining expression to get $(\lambda y.(v\ y)\ u) \to (v\ u)$.

To summarize, we have covered the grammar and operational semantics for λ-calculus. You should be feeling a mixture of confusion and comfort, if this topic is new to you. Admittedly, this doesn't directly tell us how λ-calculus is related to computation, but we will get there. One step at a time.

Next, we will discuss the grammar and operational semantics for π-calculus.