Introduction to Design of Computational Calculi 1: Mathematical Models of Grammar

0.1 Overview

This series of posts will follow Greg Meredith's Introduction to the Design of Computational Calculi (DoCC) lectures and attempt to explain the information covered. Go here to watch the first video.

To design computational calculi for specific applications, we must first understand what constitutes a computational calculus. The RhoVM is based on the concurrent computational calculus known as the reflective, higher-order calculus, or ρ-calculus for short. We will investigate the ρ-calculus and its connection to Rholang along with more classical examples of computational calculi in this series. The foundational example being that of the λ-calculus, which we will discuss in detail. This is not intended to be a thorough investigation so much as a concerted effort to convey a clear picture of what's going on with computational calculi and how they apply to blockchain technology.

To study computational calculi, we will first consider mathematical models of grammar. We discuss mathematical models for the grammar of λ-calculus and ρ-calculus in this post. Preliminary material about λ-calculus, π-calculus, and ρ-calculus can be revisited by following the appropriate link.

1 Mathematical models of grammar

We have seen in the primer series that the presentation of a computational calculus requires:

1. Grammar: rules for creating terms
2. Structural equivalence: notion of equivalence of terms
3. Operational semantics: notion of computation

We will talk a little bit about how calculi relate to other mathematical and computational structures. Specifically, we will discuss how a presentation like this sets up a structure/function relationship which is a biological idea.

The grammar of a computational calculus lives inside the BNF presentation. BNF grammars are the tool of choice when writing a parser. Within the Chomsky hierarchy, BNF grammars are essentially equivalent to context-free grammars. This gives us a way to describe structural relationships, how to build data types. The BNF grammar is the structure of our domain of discourse and tells us how the things we will use to compute with are built.

BNF grammars are self-referential, terms are built from terms recursively.

1.1 Lambda calculus

The λ-calculus models the notion of a function. Within λ-calculus, we have a first-class notion of function and pass functions around like values. This is a powerful and appealing notion for programming languages; use functions as inputs and outputs.

The grammar discussed in this series will always be presented in Baukus-Naur form (BNF) which is typical for programming languages. BNF is just a fancy way of describing how the rules for the syntax work in a grammar, i.e. what "admissible" terms look like. As an example, here is the BNF presentation of λ-calculus:

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

There are *three* term constructors which correspond to three different rules for constructing terms. This presentation gives rise to a *language*, the collection of λ-terms. Our language consists of terms M and N which are of the form x or $\lambda x.M$ or MN or some admissible combination of these. Terms are built recursively.

We will always think of terms as a computer program. Only admissible terms will represent programs that compile. Hence, x alone does not compile, but $\lambda x.x$ does.

We may generate terms from by using the constructors in the BNF presentation or we may be given a term and check whether it is a λ-term by deconstructing it.

Recall that x is just a *name* or *variable* which is supplied to us in a set X and that this grammar is *parametric* in the set of variables X. So we write

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

to emphasize the dependence of the admissible terms $M[X]$ on X.

There is only one *type* in λ-calculus: the λ-term.

1.1.1 Mathematical model

With mysterious objects, like the BNF grammar, it can be helpful to make a mathematical model.

Since λ-terms have three basic forms (x from the set X, $\lambda x.M$ where $x \in X$ and M is any λ-term, and MN where M and N are any λ-terms), the *domain equation* for λ-calculus has three terms, each corresponding to one of these basic forms.

Let's denote the set of all λ-terms by $M[X]$. Now to get a variable x, we simply choose one from the set X, i.e. we only need the set X. To form an abstraction $\lambda x.M$, we need a variable from X and a term from $M[X]$, i.e. we need the set $X \times M[X]$ (*pairs* of elements from X and $M[X]$), where

$$A \times B = \{(a, b) : \ a \in A, b \in B\}$$

is called the *direct product* of the sets A and B. To get a term like MN, we need two λ-terms M and N (both from $M[X]$), i.e. we need the set $M[X] \times M[X]$ (pairs of elements from $M[X]$). From all of this, we get the following equation describing the language emerging from this grammar

$$M[X] = X + (X \times M[X]) + (M[X] \times M[X]) \tag{1.2}$$

which we call the *domain equation*. Essentially, this just says that the λ-terms are formed from mentioning variables (which depends only on X), abstractions of a variable and a λ-term (which depends on $X \times M[X]$), and applications of two λ-terms (which depends on $M[X] \times M[X]$).

The domain equation can be interpreted in terms of sets with recursion or categories. For now, we interpret the collection of λ-terms as a set $M[X]$ which satisfies the domain equation (1.2). There is a bit of subtlety present, let me explain. The + operation in the domain equation is not ordinary addition. We are dealing with sets so, here, + means *disjoint union*. We take the union of sets and treat all elements as distinct. We think of each element as being tagged by the set it's a member of.

For example, we can think of the disjoint union, $A+B$, of the sets $A = \{a, b, c\}$ and $B = \{a, d\}$, as first, tagging the elements of A and B

$$A' = \{a_A, b_A, c_A\}, \quad B' = \{a_B, d_B\},$$

then taking the union of the tagged elements $A' \cup B' = \{a_A, b_A, c_A, a_B, d_B\} = A + B$. On the other hand, the ordinary union is $A \cup B = \{a, b, c, d\}$. No distinction is made between the element a coming from the set A and the one from B. The point of the disjoint union is that all elements are treated as distinct and thus no elements are discarded due to repetition.

The use of tagging elements in the disjoint union is synonymous with case classes in Scala. We can use our mathematical model to get a computational interpretation in Scala. We use the object-oriented features to model the features of the grammar.

Terms appearing on the left-hand side of the BNF presentation (1.1) are called *non-terminal* symbols and terms only appearing on the right-hand side are called *terminal* symbols. All of the elements are effectively an extension of a single, overarching trait. This trait is a type, $M[X]$, which is parametric in another type X. If we are given a variable, of type X, then we can form a new type, $M[X]$.

Scala allows us to create case classes to disambiguate terms just like with the disjoint union. In Scala, we define the trait $M[X]$ with three case classes:

```
trait M[X]
case class Mention[X]( x : X ) extends M[X]
case class Abstraction[X]( x : X, m : M[X] ) extends M[X]
case class Application[X]( m : M[X], n : M[X] )
```

It is interesting to note that equation (1.2) may be taken literally with X being treated as a constant and $M[X]$ as a variable which we will even differentiate with respect to! But we will save that discussion for a future post.

Now we continue on to discuss the grammar underlying Rholang and RChain.

1.2 Rho calculus

1.2.1 Mathematical model

Recall the BNF presentation of a grammar for ρ-calculus

$$\begin{aligned} P, Q &::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(Q) \mid P|Q \mid {}^*x \\ x, y &::= \ @P \end{aligned} \tag{1.3}$$

Let $P[X]$ denote the admissible terms in this grammar generated from the set of names X, i.e. using only the first set of production rules, forgetting about reflection. Denote the collection of all

ρ-calculus processes by R, i.e. using reflection. The grammar has the domain equation

$$\begin{aligned} P[X] &= 1 + (X \times X \times P[X]) + (X \times P[X]) + (P[X] \times P[X]) + X \\ R &= P[R] \end{aligned} \tag{1.4}$$

which is actually *not* parametric in the set of names X because of the recursion present in the second equation, which is called "tying the recursive knot". Let's walk through the construction of (1.4) from the productions in (1.3).

- 0
  The stopped process is a single process, hence it gives rise to a set with a single element denoted by 1. Category theoretically, 1 is the initial object in the category.

- $\text{for}(y \leftarrow x)P$
  A receive requires two names $x, y \in X$ and a process $P \in P[X]$. Hence, we need to the set $X \times X \times P[X]$.

- $x!(Q)$
  A send requires a name $x \in X$ and a process $P \in P[X]$. Hence, we need to include the set $X \times P[X]$.

- $P|Q$
  Parallel composition requires two processes $P, Q \in P[X]$. Hence, we need to include the set $P[X] \times P[X]$

- ${}^*x$
  A dereference requires only one name $x \in X$. Hence, we also need to include the set X in the domain equation.

- $x ::= @P$
  This part of the grammar arguably makes the most important contribution to the domain equation. It simply says that names are quoted processes. This means that to get the name $@P$, we require a process $P \in P[X]$. Notice the self-reference here; this is the *reflective* part of ρ-calculus. The equation $R = P[R]$ says that the set of ρ-calculus processes, R, is the recursive fixed point of the first equation, i.e. R is the set of processes generated over the set of quoted processes.

Notice in the BNF presentation (1.3) that there are two separate definitions present. The reason is because there are two *types* in ρ-calculus: processes and names. There are no variables present, hence this is a kind of machine code which means that we can directly encode processes as numbers to get a binary representation for each process in ρ-calculus.

1.2.2 Summary

The BNF presentation of a grammar gives rise to a mathematical model via the domain equation which can be interpreted as a set with recursion or a category. We constructed the domain equations for grammars for λ-calculus and ρ-calculus.

In the next post, we will discuss free and bound names and equivalence of processes.