Intro to Design of Computational Calculi 3: Program Equivalence and Operational Semantics

0.1 Overview

In the previous post in this series, we covered names, compilable programs, and equivalence of processes.

In this post, we will explain program equivalence and operational semantics. We also discuss the concepts of monoids and monads and their connection to computational calculi.

0.2 Domain-specific languages

Designing a computational calculus for use in a specific setting is synonymous with creating a domain-specific language. Domain-specific languages are an approach to problem-solving from a language perspective (e.g. HTML for web pages or Scheme). They are the product of language-oriented programming and among the most powerful problem-solving methodologies. In the blockchain space, we need a domain-specific language which captures the range of behaviors of users within the application.

0.3 Another look at alpha equivalence

The following "Java" programs execute in exactly the same way despite referencing variables with different names.

Program 1	Program 2

class C {
T m(A a){
if pred(a){
}
else{
}
}
}
	class C {
	T m(A b){
	if pred(b){
	}
	else{
	}
	}
	}

Looking at Program 1, inside the class, C, there is a method, m, that takes in an argument, a, of type A, and returns something of type T. Inside the body, we test a predicate of the argument a. If true, we do something, and if false, we do something else. Looking at Program 2, inside the class, C, there is a method, m, that takes in an argument, b, of type A, and returns something of type T. Inside the body, we test a predicate of the argument b. If true, we do something, and if false, we do something else.

We disregard the textual differences and treat these programs as the same because they execute in the same way (that's what's really important anyhow). The exact variable name does not matter as long as we are consistent with its usage. This is one way to think about α-equivalence.

Another way to think about the equivalence of programs is in terms of contexts. We say two programs are *equivalent* if they behave the same in all contexts, i.e. we can freely substitute one for the other in all contexts.

0.4 Universal algebra: monoids & monads

From the viewpoint of universal algebra, in the π- and ρ-calculi, the collection of processes together with the par operator $|$, the stopped process 0, and the equations from structural equivalence, form an algebraic structure known as a monoid.

We define a *monoid* as a set M equipped with a binary relation $* : M \times M \to M$ (i.e. for all $a, b \in M$, $a * b \in M$) and a distinguished element $e \in M$ (called the *identity*) which satisfies the following properties:

1. For all $a, b, c \in M$, we have $(a * b) * c = a * (b * c)$. (associativity)
2. For all $a \in M$, we have $a * e = a = e * a$. (left and right identity)

We will denote all of this by displaying it as $(M, *, e)$.

The monoid $(\mathcal{P}, |, 0)$ formed by the processes has the set of processes $\mathcal{P}$ as the underlying set, the par operator $|$ as the binary relation, the stopped process 0 as the identity, and the laws for structural equivalence make $|$ associative and 0 the identity. We actually have additional structure in this case since the order in which we par processes together is irrelevant. So $(\mathcal{P}, |, 0)$ is, in fact, a *commutative* monoid.

Bill Lawvere identified a relationship between the grammar-style presentation and the monadic presentation in the 1960's. He showed that the grammar-style presentation can be captured in what is called a *Lawvere theory* and that Lawvere theories have a correspondence (isomorphism) to a monad, i.e. you can either start with the monad and get the theory or vice versa. Structural equivalences correspond to the algebras of the monad.

In the 2000's, Jamie Gabbay and Andy Pitts showed that the grammar of a computational calculus, in most cases, is representable by a monad, and used the monadic machinery to abstract out the variable machinery.

When we generate type systems using LADL, the monad will represent a key ingredient. Structural equivalence is the other ingredient; a wide range of monads give us freely-generated structures, a lot like the language generated from a grammar with no notion of equivalence of terms.

0.4.1 Example

One can freely generate the set of all strings over an alphabet $A = \{a, b\}$. This collection, $L(A)$, is known as the *language* over the alphabet A. We can do better, however. This language can be interpreted as a monoid with $\emptyset$ as the empty string using concatenation, $*$, as the binary operation. Since string concatenation is naturally associative and the empty string serves as the identity. For example,

$$a * b = ab,\ aa * abba = aaabba,\ bbaa * \emptyset = bbaa = \emptyset * bbaa$$

We can transform this monoid into a *group* by adjoining complementary elements $\bar{a}$ and $\bar{b}$ satisfying the equations

$$a * \bar{a} = \bar{a} * a = \emptyset = \bar{b} * b = b * \bar{b}$$

This makes it so that each element of $L(A)$ has an inverse, i.e. for each string s_1, there is a string s_2 such that $s_1 * s_2 = s_2 * s_1 = \emptyset$. For example, the inverse of $ab\bar{a}\bar{a}\bar{b}$ is $baa\bar{b}\bar{a}$ (why?).

1 Operational semantics i.e. reduction rules

1.1 Lambda calculus

The rule for β-reduction in λ-calculus applies when an abstraction, which binds a variable in a term, is immediately followed by an application at the top level.

$$(\lambda x.MN) \to M\{N/x\}$$

1.2 Rho calculus

The *comm rule* for reduction in ρ-calculus applies when a send and a receive on the same channel are run in parallel.

Comm rule:

$$\text{for}(y \leftarrow x)P \mid x!(Q) \to P\{@Q/y\}$$

The comm rule has the following interactions with the par operator and structural equivalence.

Par rule:

$$\text{If } P \to P', \text{ then } P \mid Q \to P' \mid Q.$$

Structural equivalence rule:

$$\text{If } P \equiv P',\ P' \to Q', \text{ and } Q' \equiv Q, \text{ then } P \to Q.$$

Unlike the laws for structural equivalence, where information is preserved, through reduction, structure is lost or forgotten. Notice that the right side of the comm rule no longer refers to the name x.

1.2.1 Example

The reduction rules are purposely left in a succinct format because we can combine them at will. Say we have the expression

$$\text{for}(y \leftarrow x)P|R|x!(Q)$$

We notice that there is a send and a receive on the channel x so there will be a comm event. However, we can not immediately reduce this expression because the comm rule does not apply as is. We must first put the send and receive next to each other. Luckily, because of the structural equivalence laws, we know the par operator is commutative. Hence, we can change the order in which processes are par-ed together free of charge. So we switch the order of R and $x!(Q)$ to get

$$\text{for}(y \leftarrow x)P|x!(Q)|R$$

Now that the send and receive are adjacent, they can reduce under the comm rule, or can they?... The process R is also present. The par rule comes to the rescue. Since $\text{for}(y \leftarrow x)P|x!(Q) \rightarrow P\{@Q/y\}$, we have

$$\text{for}(y \leftarrow x)P|x!(Q)|R \rightarrow P\{@Q/y\}|R$$

Putting all the pieces together with the structural equivalence rule, we get

$$\text{for}(y \leftarrow x)P|R|x!(Q) \rightarrow P\{@Q/y\}|R$$

1.3 Summary

We covered the notions of program equivalence and operational semantics. We also discussed how a language can be interpreted as a monoid and how the grammar-style presentation of a language relates to a Lawvere theory and hence a monad. Since monads are generalizations of monoids, we presented the definition of a monoid. Lastly, we discussed the operational semantics for λ-calculus and ρ-calculus.

The next post in the DoCC series will discuss the injection of names into rho calculus.