Introduction to Design of Computational Calculi 2: Names, Compilable Programs, and Equivalences

0.1 Overview

We discuss the role of names and binding and the different notions of equivalence in our examples of computational calculi. Keeping in mind that the goal of these lectures is to be able to design computational calculi with desired properties, we discuss these ideas generally, yet within the framework of our examples.

0.2 Process calculi

The distinguishing feature of a mobile process calculus is that the agents perform computation solely by passing messages. This is quite different from λ-calculus, where computation is performed by manipulation of a shared set of variables.

There are two types of process calculi: synchronous and asynchronous. The distinguishing feature being that in an asynchronous process calculus, we may never know if a sent message is received, i.e. we do not have a notion of synchronization. The variant of Robin Milner's π-calculus presented in the primer is synchronous, but can be made asynchronous. Other asynchronous process calculi include ambient calculus, blue calculus, join-calculus, and ρ-calculus.

0.3 Presentation of a collection

In previous posts, we discussed how the BNF grammar acts as a recipe for both recognizing terms of the language and creating terms in the language. There are two distinct ways of presenting a collection of objects, such as the terms of a calculus: extensional and intensional.

An *extensional* presentation of a collection explicitly lists the elements. For example, an extensional presentation of the natural numbers is given by $\{0, 1, 2, 3, 4, ...\}$. However, if we are dealing with very large collections, possibly infinite, an extensional presentation may be awkward or even worse, impossible.

An *intensional* presentation of a collection implicitly defines the collection by inclusion based on satisfaction of a property. For example, we have the intensional presentation of the odd integers

$$\{n \in \mathbb{Z} : n = 2k + 1 \text{ for some } k \in \mathbb{Z}\}.$$

This simply says that given an integer n, we can test to see if it is odd by trying to find an integer k satisfying the relationship $n = 2k + 1$. If such an integer k exists, then n belongs to the collection of odd integers, and if not, then n is not an odd integer.

We may use a one-line Haskell program to generate the natural numbers by utilizing an intensional presentation. An intensional presentation exploits information theoretic regularities in the collection to present it in a succinct way.

In the context of computational calculi, the grammar is an intensional presentation of the terms in the language. The grammar provides constraints, but terms are freely generated, i.e. they can be comprised of arbitrarily many sub-terms.

1 Names and equivalences

Since names play a prominent role in process calculi, providing channels to send messages on and the messages themselves in some cases, and all computation is done through message-passing, we must analyze the names in our calculi more closely. We will start with the definition of free names in λ-calculus to help build our intuition about ρ-calculus.

1.1 Free and bound names in lambda calculus

Recall that there are three term constructors in λ-calculus; x (*mention*), $\lambda x.M$ (*abstraction*), (MN) (*application*). We single out abstraction because the term $\lambda x.M$ *binds* x *in* M or more generally, λ is a *binder*, it marks x as a variable in M. Using this abstraction, it now becomes possible to substitute something for x in M, this is of course done via β-reduction. Since all λ-terms are ultimately built from names and some names are bound by abstraction, we would like to make formal the notion of bound and free names in a λ-term. We define the *free* names, $\mathcal{FN}(M)$, of a λ-term, M, recursively by defining the free names for each term constructor in λ-calculus:

- $\mathcal{FN}(x) = \{x\}$
  The only free name in a mention is the name mentioned.
- $\mathcal{FN}(\lambda x.M) = \mathcal{FN}(M) \setminus \{x\} = \{n \in \mathcal{FN}(M) : n \neq x\}$
  Abstractions turn free names into bound names, hence the abstracted name x is discarded from the set of free names in the abstracted term M. This is the general action of a binder. (For those not familiar with the set minus notation $A \setminus B$, it simply means discard the elements of B from A.)
- $\mathcal{FN}((MN)) = \mathcal{FN}(M) \cup \mathcal{FN}(N)$
  The free names in an application are the union of the free names in the terms being applied.

Further, we let $\mathcal{N}(M)$ denote the collection of names mentioned in the term M. A name x in M, $x \in \mathcal{N}(M)$, is called *bound* if it is not free. Denoting the collection of bound names in M by $\mathcal{BN}(M)$, we have the relationship

$$\mathcal{N}(M) = \mathcal{BN}(M) \cup \mathcal{FN}(M)$$

where this is, in fact, a disjoint union (why?).

A *closed term* is a term with no free names, i.e. M is a closed term if and only if $\mathcal{FN}(M) = \emptyset$ where $\emptyset = \{\}$ denotes the empty set, the set containing no elements. Closed terms correspond to compilable programs. There is also a notion of a name being fresh in a term. To say that a name x is *fresh* in a term M simply means that $x \notin \mathcal{N}(M)$. Therefore abstraction of the name x has no effect on M.

1.1.1 Examples

(i) $\lambda x.x$ is a closed term and $\mathcal{FN}(\lambda x.x) = \{x\}$
This should be obvious considering that we are abstracting the only name mentioned. Nonetheless, we do the computation carefully using the above rules to verify our intuition.

$$\mathcal{FN}(\lambda x.x) = \mathcal{FN}(x) \setminus \{x\} = \{x\} \setminus \{x\} = \emptyset$$

(ii) $\lambda x.\lambda y.x$ is a closed term and $\mathcal{N}(\lambda x.\lambda y.x) = \{x\}$
Again, we have abstracted the only name mentioned x, but now we also have an additional abstraction of a fresh name y. The term is still closed since

$$\mathcal{FN}(\lambda x.\lambda y.x) = \mathcal{FN}(\lambda y.x) \setminus \{x\} = (\mathcal{FN}(x) \setminus \{y\}) \setminus \{x\} = (\{x\} \setminus \{y\}) \setminus \{x\} = \{x\} \setminus \{x\} = \emptyset$$

(iii) $\lambda x.(x\ y)$ is not a closed term; $\mathcal{FN}(\lambda x.(xy)) = \{y\}$ and $\mathcal{N}(\lambda x.(x\ y)) = \{x, y\}$
This is straightforward considering that two names, x, y, are mentioned in the term, but only one, x, is bound by abstraction. Explicit calculation shows

$$\mathcal{FN}(\lambda x.(x\ y)) = \mathcal{FN}((x\ y))\backslash\{x\} = \mathcal{FN}(x) \cup \mathcal{FN}(y) \backslash \{x\} = \{x\} \cup \{y\} \backslash \{x\} = \{x, y\} \backslash \{x\} = \{y\}$$

Let's discuss how these free and bound names effect the terms.

1.2 Alpha equivalence

For a free name $x \in \mathcal{FN}(M)$, subsequent abstraction (binding) and substitution of a fresh name y in M results in the α-conversion

$$\lambda x.M \to_\alpha \lambda y.M\{y/x\}$$

Two terms M, N are called α-equivalent, denoted by $M \equiv_\alpha N$, if one α-converts to the other.

- α-equivalence erases obvious syntactic distinctions which make no difference for computation, i.e. different names are used for variables, but the programs do the same thing.
- Semantic equivalence for syntactic distinctions. The symbols in the terms are literally different, but the same meaning is given to the terms. This is the distinction between *sense* and *denotation*; what is written and what is meant by what is written.

Introducing binding operators into a computational model shifts it from being an algebra to a calculus because we may then define a notion of *reduction*, i.e. β-reduction in λ-calculus and the *comm rule* in π-calculus and ρ-calculus.

Now we switch our focus to ρ-calculus.

1.3 Free and bound names in rho calculus

In ρ-calculus, we also define the free names, $\mathcal{FN}(P)$, in a process, P, by defining the free names for each term constructor:

- $\mathcal{FN}(0) = \emptyset$
  The stopped process has no free names, in fact, no names at all.

- $\mathcal{FN}(\text{for}(y \leftarrow x)P) = (\mathcal{FN}(P) \setminus \{y\}) \cup \{x\}$
  In a receive, the name being listened for, y, is bound and the channel listened on, x, constitutes a free name. Importantly, the name y we listen for is bound in the process P so we discard y from the collection of free names in P, hence the appearance of $\mathcal{FN}(P) \setminus \{y\}$. Then x is added to this collection. For this reason, the for-comprehension is a binder in ρ-calculus.

- $\mathcal{FN}(x!(P)) = \mathcal{FN}(P) \cup \{x\}$
  In a send, the channel x the message $@P$ is sent on is a free name, along with the free names in the process P being sent.

- $\mathcal{FN}(P|Q) = \mathcal{FN}(P) \cup \mathcal{FN}(Q)$
  Analogously to application in λ-calculus, the collection of free names in the parallel composition of two processes is the union of the collections of free names in each process.

- $\mathcal{FN}(^*x) = \{x\}$
  The only free name in a dereference $^*x$ is the name being dereferenced x. In this context, there is no access to the names in the underlying process of x. This is a very important, yet subtle, point. Since each name x has an underlying process P, i.e. $x = @P$, we can rightfully think of the process $^*x$ as $^*@P$ and wonder how it relates to the original process P. Notice that $\mathcal{FN}(^*@P) = \{@P\}$, i.e. we do not have access to the free names in P, only the name $@P$, at this level.

1.4 Structural congruence a.k.a. structural equivalence

Structural congruence is a semantic equivalence for syntactically distinct processes. For example, the process $P|Q$ denotes "P running in parallel with Q" which means the same thing as "Q running in parallel with P," denoted by $Q|P$. Thus, we make these processes equivalent with the relation $P|Q \equiv Q|P$, i.e. the semantics impose commutativity of parallel composition.

Recall, structural equivalence for ρ-calculus, written $\equiv$, is the least congruence, containing α-equivalence and satisfying the laws

$$P|0 \equiv P, \quad P|Q \equiv Q|P, \quad P|(Q|R) \equiv (P|Q)|R$$

1.5 Name equivalence

In ρ-calculus, processes can be turned into names by quoting and names can be turned into processes by dereference. Since names play a role distinct from processes, we also define the notion of *name equivalence*, written $\equiv_N$, as follows:

- $@(^*x) \equiv_N x$
  If we start with the name x, dereference it to the process $^*x$, and then quote this process to the name $@(^*x)$, this name is name equivalent to the original x.

- If $P \equiv Q$, then $@P \equiv_N @Q$
  If the processes P and Q are structurally equivalent, then the names $@P$ and $@Q$ are name equivalent.

1.6 Connections to other concepts

Universal algebra is study of general algebraic structures through the constructions of generators and relations. Generators give us a starting point and relations give us a notion of equivalence of elements in much the same way that in a computational calculus, the grammar gives us a starting point and the structural equivalence is our notion of equivalence. The case of a computational calculus turns out to give a deeper articulation of the generators and relations than that presented in universal algebra.

There are so many connections between computational calculi and category theory that we could write several posts about these connections alone. We will cover some of these category theoretic connections in later posts. For now, it suffices to mention the analogous roles in category theory of free monads to grammars and algebras on free monads to structural equivalence.

Rho calculus (and reflection, in general) opens up an entire universe of set theories based on the notion of annihilation used. We will discuss this further in later posts.

1.7 Summary

We discussed the notions of free and bound names in λ-calculus and ρ-calculus and how this relates to compilable programs. We also covered the different notions of equivalence used in ρcalculus: α-equivalence, structural equivalence, and name equivalence.

There are many connections to other areas of mathematics, notably universal algebra and category theory, and computer science which we have only begun to discuss.

In the next post, we expand on our discussion of operational semantics.