Computational Calculus Primer Part 3: Rho calculus

0.1 Overview

We saw in the previous post that π-calculus is a process calculus which is dependent on a theory of names; it is therefore said that it is not a closed theory. The ρ-calculus is an asynchronous message-passing calculus built on a notion of quoting; it is a closed theory, as the theory of names is wholly determined by the theory of processes. The name ρ-calculus or RHO-calculus is an acronym for *reflective, higher-order calculus*. By reflective, we mean there is a mechanism for switching between names and processes (quote/dereference, see below). By higher-order, we mean that the resulting laguage is very expressive.

This is the last post in the primer series. The upcoming posts will follow the DoCC lectures.

1 Rho calculus

Coming from π-calculus, ρ-calculus can seem a little like stepping into the twilight zone. In π-calculus, names and process were two distinct dichotomies. However, in ρ-calculus, we have constructors for turning a process into a name and vice versa. The thinking is that processes are programs and since names are quoted processes, names represent the code of a process. Therefore, one can run a process, i.e. execute the program, quote the process to bundle up its code, send the code to another channel, and dereference the code to get a running instance of the program.

As in π-calculus, in ρ-calculus we represent interaction between agents by message-passing.

1.1 Grammar

The BNF presentation of a grammar for ρ-calculus, written in the style of Rholang,

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$

$$x, y \ ::= \ @P$$

has several similarities with π-calculus. The stopped process, send, receive, and parallel composition all have analogues in ρ-calculus. However, it is the differences that set it apart. The ability to quote a process into a name and deference a name into a process enhances the expressivity of the ρ-calculus.

- The stopped process: 0

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$

$$x, y \ ::= \ @P$$

This is the atomic process that means “do nothing”. All of the terms in the ρ-calculus are built from this special process in a similar way to how the terms in the λ-calculus and π-calculus are built from names. The main difference being that we no longer require a set of names to get the ball rolling with our term formation. The stopped process is enough to do the job.

- For-comprehension: $\text{for}(y \leftarrow x)P$

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$
$$x, y \ ::= \ @P$$

  This process is similar to an input-guarded continuation, $x(y).P$, in π-calculus. This process waits to receive a message on channel x, then the process P will run with the received name substituted for the name y. See the comm rule below.

- Send on a channel x: $x!(P)$

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$
$$x, y \ ::= \ @P$$

  The quoted process $@P$ is sent on the channel x. This is analogous to output prefixing, $\bar{x}\langle y\rangle.P$, in π-calculus, except now we can reinstate our messages as processes using dereference.

- Parallel composition of processes: $P \mid Q$

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$
$$x, y \ ::= \ @P$$

  The processes P and Q run concurrently, exactly as in π-calculus.

- Dereference (“name drop”) or dequote: ${}^*x$

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$
$$x, y \ ::= \ @P$$

  ${}^*x$ - read “drop x” - allows us to turn a name back into its underlying process (since each name is a quoted process). To turn the name $@P$ into its underlying process, derefernce it.

- Names are quoted processes: $@P$

$$P, Q \ ::= \ 0 \mid \text{for}(y \leftarrow x)P \mid x!(P) \mid P|Q \mid {}^*x$$
$$x, y \ ::= \ @P$$

  Every name is a quoted process. The idea is that names are computer code and processes are computer programs; code is static, programs execute. Given a program (process), we can write down its code (quote it), and once we have the code (name is a quoted process), we can execute the program (dereference it). Therefore, we will take the process ${}^*@P$ to be the same as the process P. This is a consequence of our *semantic substitution*.

We are given the stopped process, 0, to start with. Out of this process alone, we construct all names and processes in ρ-calculus.

Let's build several simple ρ-calculus processes before moving on to structural congruence, other notions of equivalence, syntactic and semantic substitutions, and operational semantics.

- 0
  This is the simplest process. Nothing happens.
- $*0$
  This is *not* proper syntax - only names can be dereferenced and 0 is a process, not a name. This is not a valid ρ-calculus process!
- $0 \mid 0$
  This is arguably the second simplest process; it concurrently runs the stopped process with itself. Should we be able to simplify this process?
- $*@0$
  Since 0 is the simplest process, @0 is the simplest name. This process dequotes the name @0. Should we be able to simplify this process?
- $@(0 \mid 0)!(0)$
  Since $0 \mid 0$ is a process, $@(0 \mid 0)$ is a name which provides a channel to send messages on. This process sends the message @0 on the channel $@(0 \mid 0)$.
- $\text{for}(@(0 \mid 0) \leftarrow @0)0$
  This process waits for a message on the channel @0. When it is received, it substitutes the received name for $@(0 \mid 0)$ in the process 0.
- $@0!(0) \mid \text{for}(@0 \leftarrow @0)0$
  This process concurrently sends and receives the message @0 on the channel @0. Just like in π-calculus, this will result in a communication!
- $@(@0!(0))!(0)$
  Since $@0!(0)$ is a process, we can quote it to form the name $@(@0!(0))$ which gives us another channel to send messages on. This process sends the name @0 on the channel $@(@0!(0))$.
- $\text{for}(@0 \leftarrow @(0 \mid 0))@0!(0)$
  This process waits for a message on the channel $@(0 \mid 0)$ to substitute for the name @0 in the process $@0!(0)$.

Now we will discuss the different notions of equivalence in ρ-calculus.

1.2 Notions of equivalence

1.2.1 Structural congruence

Just like in π-calculus, we expect some of the processes to be equivalent from the viewpoint of computation. Again, this equivalence, denoted by $\equiv$, is called *structural congruence*. This is the least equivalence, containing α-equivalence that satisfies the following laws

- $P \mid 0 \equiv P$
- $P \mid Q \equiv Q \mid P$
- $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

Structural congruence only pertains to processes.

1.2.2 Name equivalence

Name equivalence, denoted by $\equiv_N$, has two instantiations pertaining to (i) a single name and (ii) structurally congruent processes:

(i) $@({}^*x) \equiv_N x$
This equivalence says quoting a dereferenced name is name equivalent to the name itself. The thinking is that if we turn some code into a program and turn that program into its code, then we should reproduce the original code.

(ii) If $P \equiv Q$, then $@P \equiv_N @Q$.
This equivalence says that structurally congruent processes, P and Q, have name equivalent quotations, $@P$ and $@Q$. The idea is that two programs which have equivalent execution also have equivalent code.

1.2.3 α-equivalence

Recall the concept of α-conversion in λ-calculus. It said two terms were equivalent if they only differ by their bound names, i.e. they only differ by what we call the variables in the term. There is an analogous concept called *α-equivalence*, denoted by $\equiv_\alpha$, in ρ-calculus (and also π-calculus). Two ρ-calculus processes are called α-equivalent if they differ only by what they call the names they reference. That is, $P \equiv_\alpha Q$ if P and Q only differ by the names used to construct these processes. A few examples of α-equivalent processes

$$x!(0) \equiv_\alpha y!(0),$$

$$P \mid \text{for}(y \leftarrow x)0 \equiv_\alpha P \mid \text{for}(w \leftarrow u)0$$

Two α-equivalent processes have the same code so they both represent the same program.

1.3 Substitutions & operational semantics

As in other computational calculi, we have rules for substitutions and what these substitutions mean in terms of computation. We will denote the substitution of the name y for the name x within the process P by $(P)\{y/x\}$.

1.3.1 Syntactic substitution

The rules for *syntactic substitutions* dictate how the usual substitution of names translates to that of processes (because processes are built from 0 and names). On the level of names, substitution is straight forward, but we must build the concept of name substitution in a process. Thus, we have a rule for each term constructor. The rules for name substitution in processes are built from:

- $(0)\{@Q/@P\} = 0$
  Any substitution of names in the stopped process does not change the stopped process;

- $(!R)\{@Q/@P\} = !(R)\{@Q/@P\}$
  There is only one process to do substitution with in replication and it proceeds in the obvious way;

- $(R \mid S)\{@Q/@P\} = (R)\{@Q/@P\} \mid (S)\{@Q/@P\}$
  Substitution in processes running in parallel is just substitution in each process separately and then we run these new processes in parallel;

- $(x!(R))\{@Q/@P\} = (x)\{@Q/@P\}!((R)\{@Q/@P\})$
  Substitution in a send works as expected. First, we do the appropriate substitution with channel name x; this explains the appearance of $(x)\{@Q/@P\}$. Then, we do the appropriate substitution with the process R; this explains the appearance of $(R)\{@Q/@P\}$;

- $(\text{for}(y \leftarrow x)R)\{@Q/@P\} = \text{for}(z \leftarrow (x)\{@Q/@P\})((R)\{z/y\})\{@Q/@P\}$
  This rule looks very complicated so let's unravel it. First, we do a name substitution with the receiving channel x; this explains the appearance of $(x)\{@Q/@P\}$ within the for-comprehension. Then, we call the message we're waiting for, y, by a different name, z. Since we switched this label, we must substitute z for every appearance of y in the process R; this explains the appearance of $(R)\{z/y\}$. Ultimately, we need to do the name substitution within the process $(R)\{z/y\}$; this explains the appearance of $((R)\{z/y\})\{@Q/@P\}$;

- $(^*x)\{@Q/@P\} = \begin{cases} ^*@Q & \text{if } x \equiv_N @P \\ ^*x & \text{otherwise} \end{cases}$
  Substitution with a dereferenced name is straight forward. If the name x is name equivalent to $@P$, then we do the expected substitution, i.e. $@Q$ is substituted for x. If x is not name equivalent to $@P$, nothing changes;

  The usual name substitution is given by

$$(x)\{@Q/@P\} = \begin{cases} @Q & \text{if } x \equiv_N @P \\ x & \text{otherwise} \end{cases}$$

In ρ-calculus, we must also take name equivalence into consideration. So we perform the substitution for any name equivalent name too.

These syntactic substitution rules tell us how to write down the substitutions for processes, but they do not necessarily tell us what these substitutions mean in terms of computation. This is where the semantic substitution comes in.

1.3.2 Semantic substitution

Some real special sauce comes from the notion of *semantic substitution.* We have discussed quoting processes and dereferencing names, how quoting a process results in a name, and how dereferencing a name results in a process, but we have not explicitly connected the two ideas.

Formally speaking, we have the rule

$$({}^*x)\{@Q/@P\} = \begin{cases} Q & \text{if } x \equiv_N @P \\ {}^*x & \text{otherwise} \end{cases}$$

which says that if we substitute the name $@Q$ for the name $@P$ in the dereference process ${}^*x$, then we get the process Q if x is name equivalent to $@P$, and just ignore the substitution altogether if x is not name equivalent to $@P$.

The takeaway is that after all substitutions are done, ${}^*@P = P$ for every process P.

This is exactly what we need to happen to ensure that we can think about processes as programs and names as code. We want to be able to take a program (process), write down its code (quote the process), then run the code as a program (dereference the quoted process) and get back the original program (process). The semantic substitution brings this all together.

1.3.3 Operational semantics

We also have the notion of a *comm event* in ρ-calculus. The idea is that we have communication when we concurrently send and receive on the same channel. Communication is the reduction of a process of the form

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

We extend this notion of communication using name equivalence. This amounts to communication being generalized to the concurrent sending and receiving of messages on name equivalent channels.

- **Comm rule:**

$$\frac{x_0 \equiv_N x_1}{x_0!(Q) \mid \text{for } (y \leftarrow x_1)P \rightarrow (P)\{@Q/y\}}$$

  This rule says that the concurrent send, $x_0!(Q)$, and receive, for $(y \leftarrow x_1)P$, on name equivalent channels, x_0 and x_1, reduces to the atomic process, $(P)\{@Q/y\}$. The communication rule makes it possible for agents to synchronize and communicate processes packaged as names.

The comm rule interacts with parallel composition and structural congruence just like it does in π-calculus:

- *Parallel composition:* $\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$
  This rule says that if a process P reduces to the process P', then the process $P \mid Q$ will reduce to the process $P' \mid Q$.

- *Structural congruence:* $\dfrac{P \equiv P',\ P' \rightarrow Q',\ Q' \equiv Q}{P \rightarrow Q}$
  This rule says that if we have a process P which is structurally congruent to the process P', a process Q which is structurally congruent to the process Q', and the process P' reduces to the process Q', then the process P reduces to the process Q.

It's time for a few examples of reduction.

1.3.4 Examples

1. $@0!(0) \mid \text{for}(@(@0!(0)) \leftarrow @0)0 \;\rightarrow\; 0$
   The send, $@0!(0)$, and receive, $\text{for}(@0 \leftarrow @0)0$, processes run concurrently on the same channel, $@0$. Thus, a comm event occurs and this process reduces to the substitution
   $$(0)\{@0/@(@0!(0))\}.$$
   Recall that $(0)\{z/y\} = 0$ for any names y, z, thus the process ultimately simplifies to the stopped process 0.

2. $\text{for}(@0 \leftarrow x)@0!(0) \mid x!(@0!(0)) \;\rightarrow\; @(@0!(0))!(0)$
   First, we notice that we are sending and receiving messages on the same channel x so communication occurs. To simplify matters, let's make $P = @0!(0)$ and $Q = @0!(0)$ so the process we are dealing with can be written
   $$\text{for}(@0 \leftarrow x)P \mid x!(Q).$$
   Using the comm rule (here $y = @0$), this reduces to
   $$(P)\{@Q/@0\}.$$
   Using the definitions of P and Q, we have
   $$(@0!(0))\{@(@0!(0))/@0\}$$
   and using the syntactic substitution rule for a send process, we substitute in the name and the process, to get
   $$(@0)\{@(@0!(0))/@0\}!((0)\{@(@0!(0))/@0\}).$$
   Again using the rules of syntactic substitution, the expression simplifies to
   $$@(@0!(0))!(0).$$
   Therefore, the original process ultimately reduces to a send of $@0$ on the channel $@(@0!(0))$.

3. $\text{for}(@0 \leftarrow @(@0!(0)))^*(@0) \mid @(@0!(0))!(!(@0!(0))) \;\rightarrow\; !(@0!(0))$
   First, let's pin down exactly what is going on here. There are two processes running concurrently, the receive process, $\text{for}(@0 \leftarrow @(@0!(0)))^*(@0)$, and the send process, $@(@0!(0))!(!(@0!(0)))$. To simplify matters, let's call $y = @0$, $P = {}^*y$, $Q = y!(0)$, $x = @Q$, and $R = !Q$. Now we rewrite the original process as
   $$\text{for}(y \leftarrow x)P \mid x!(R).$$
   In this format, we clearly see that a comm event occurs; the process reduces to
   $$(P)\{@R/y\}.$$
   Going back to the definitions of P and Q, we have
   $$({}^*y)\{@R/y\}.$$
   By the semantic substitution rule for dereferencing, this substitution simplifies to the process R, which is just
   $$!(@0!(0)),$$
   replication of sending the message $@0$ on the channel $@0$.

1.3.5 References

1. Meredith, Radestock (2005). *A Reflective Higher-Order Calculus*. Retrieved June 16, 2018.