Computational Calculus Primer Part 2: Pi calculus

0.1 Overview

Our interest in π-calculus lies in the fact that it is the precursor of ρ-calculus. It is a model of concurrent computation, or more specifically, a process calculus based upon a notion of naming. In a process calculus, we represent interactions between independent *agents* or *processes* as message-passing, as opposed to modifications of shared variables (as in λ-calculus). The π-calculus differs from λ-calculus in the distinction made between names and processes and by the addition of term constructors, including *parallel composition* which is responsible for concurrency. So there are two distinct entities in π-calculus, processes and names, and different interactions between processes and names. However, just like λ-terms, the processes in π-calculus are built from names.

Parallel composition allows the computation in processes to proceed either concurrently and independently or communicate on a shared channel. This is useful for smart contract applications, storing state transitions of a virtual machine, and ultimately, for scalability.

In this post, we discuss the grammar, structural congruence, and operational semantics of π-calculus.

1 Pi calculus

Here we will discuss a slightly simplified version of π-calculus proposed by Robin Milner [1].

1.1 Grammar

The BNF presentation of a grammar for π-calculus

$$P, Q \ ::= \ 0 \mid x(y).P \mid \bar{x}y.P \mid (\text{new } x)P \mid P|Q \mid !P \tag{1.1}$$

much like that of λ-calculus, is parametric in a set of names X. Without even knowing what these operators mean, we already know that we can write

$$P[X] \ ::= \ 0 \mid X(X).P[X] \mid \bar{X}X.P[X] \mid (\text{new } X)P[X] \mid P[X]|P[X] \mid !P[X]$$

to express this dependence on X explicitly.

Returning to (1.1), the admissible terms in this grammar, i.e. the expressions on the right-hand side of this presentation, are also called *productions* or *term constructors*. We discuss these productions below.

Stopped process: 0
This process represents "do nothing", i.e. execution is complete, and it has stopped.

Input-guarded continuation or *input prefixing*: $x(y).P$
This process waits for a name sent on channel x to substitute for the name y in the process P. This plays an analogous role to abstraction of the name y, $\lambda y.P$, in λ-calculus, binding the name y in the process P.

Send on a channel and execute or *output prefixing*: $\bar{x}y.P$
Sends the name y on the channel x as a substitute name in an input-guarded process waiting on the channel x and then runs P.

Restriction or *creation of a new name*: (new x)P
$(\nu x)P$ - pronounced "new x in P" - defines x as a new name and binds it in the process P.

Parallel composition: $P \mid Q$
$P \mid Q$ - pronounced "P par Q" - means the processes P and Q are concurrently active; they can act independently, or they can communicate with one another.

Replication: $!P$
$!P$ - pronounced "bang P" - means $P \mid P \mid \ldots$; as many copies as you wish. This process can always create a new copy of P. The reduction rules make it so that there is no risk of infinite concurrent activity.

Every process in π-calculus is built from the stopped process 0. Let's build a few simple processes.

1.1.1 A few simple processes

- 0
  This is the simplest process, out of which all other processes are created.
- $x(y).0$
  This process waits for the message y on the channel x and then stops.
- $\bar{x}y.0$
  This process sends the message y on the channel x and then stops.
- (new x)0
  This process declares a new name x in the stopped process. Do you think this should correspond to a process we already have?
- $0 \mid 0$
  The stopped process runs concurrently with the stopped process. Do you think this should correspond to a process we already have?
- $\bar{x}y.0 \mid x(z).0$
  This process concurrently sends and receives a message on the channel x. This is how communication occurs.

Concurrently sending and receiving messages on the same channel allows for communication on that channel. In general, the process $\bar{x}z.Q \mid x(y).P$ corresponds to a communication on channel x. It is said that x and $\bar{x}$ are in a *name/co-name* relationship.

1.2 Structural congruence

Intuitively, we expect equivalence between some of these processes. For instance, $P \mid 0$, $0 \mid P$, and P should all be equivalent. Think about it. The process $P \mid 0$ means that we are running the process P concurrently with the stopped process 0. That shouldn't be any different than running the process P by itself. In general, we denote the *structural equivalence* of processes P and Q by $P \equiv Q$. Our intuitions are codified by the following axioms of structural equivalence.

- **α-equivalence:**
  $P \equiv Q$ if Q can be obtained from P by renaming one or more names referenced in P (we will discuss free and bound names in more depth later in the series). This is analogous to α-conversion in λ-calculus. This amounts to two programs being equivalent if they only differ by the names of variables used within. They execute the same code, but maybe calls things by different names in a consistent manner.

- **Axioms for parallel composition:**

  - $P \mid Q \equiv Q \mid P$
    The order in which we list concurrent processes shouldn't matter because they execute at the same time. Therefore, the processes $P \mid Q$ and $Q \mid P$ are equivalent.
  - $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
    The par operator should be associative. We only ever talk about "par-ing" two processes together, so if we want to throw a third process into the mix, there are two distinct ways to do that. This equivalence says these two ways are the same because we are running these processes concurrently.
  - $P \mid 0 \equiv P$
    Running the stopped process along with another process P shouldn't be any different than running P alone. This equivalence says exactly that.

- **Axioms for restriction:**

  - $(\text{new } x)(\text{new } y)P \equiv (\text{new } y)(\text{new } x)P$
    When declaring new names in a process, the order shouldn't matter. This equivalence captures that notion.
  - $(\text{new } x)0 \equiv 0$
    Declaring a new name in the stopped process shouldn't change anything. Therefore, it should be equivalent to the stopped process.

- **Axiom for replication:**

  - $!P \equiv P \mid !P$
    With replication of a process, we can always create a new copy of the process. Therefore, the processes $!P$ and $P \mid !P$ are equivalent.

- **Axiom relating restriction and parallel:**
  If the name x is bound in Q, then $(\text{new } x)(P \mid Q) \equiv (\text{new } x)P \mid Q$.

These are the properties of structural equivalence in π-calculus. Structural equivalence captures the notion of computational indistinguishably. That is, two structurally equivalent processes are identical from the point of view of computation.

Structural congruence is an example of what is called an *equivalence relation* in mathematics. We view it as computational equivalence of processes.

We will now make the notion of communication in π-calculus precise and discuss the operational semantics.

1.3 Operational semantics i.e. reduction rules

Recall that the prefix $\bar{x}y$ represents sending a message y on the channel x, and the prefix $x(z)$ represents receiving a message z on the channel x. The thinking is that when a send and receive on the same channel happen concurrently, communication takes place and the receiver then uses the passed information. In symbols, this is written

$$\bar{x}y.P \mid x(z).Q \;\rightarrow\; P \mid Q\{y/z\}$$

The $\rightarrow$ is notation for "reduces to" and we call this the *comm reduction* or *comm rule*. This means that if we find two processes $\bar{x}y.P$ and $x(z).Q$ running concurrently, then y is sent on channel x, y replaces z in the process Q, and the processes P and $Q\{y/z\}$ run concurrently. This is analogous to β-reduction in λ-calculus. It's so important, I'll write it twice.

- **Comm rule:**
$$\bar{x}y.P \mid x(z).Q \;\rightarrow\; P \mid Q\{y/z\}$$

The comm rule interacts with parallel composition, restriction, and structural equivalence.

- Parallel composition:
  If $P \rightarrow Q$, then $P \mid R \rightarrow Q \mid R$.

- Restriction:
  If $P \rightarrow Q$, then $(\text{new } x)P \rightarrow (\text{new } x)Q$.

- Structural Equivalence:
  If $P \equiv P'$, $Q \equiv Q'$, and $P' \rightarrow Q'$, then $P \rightarrow Q$.

We should get our feet wet with some examples.

1.3.1 Examples

(i) $\bar{x}y.0 \mid x(z).P \mid Q \;\rightarrow\; P\{y/z\} \mid Q$
Proof: The first two processes, $\bar{x}y.0$ and $x(z).P$, will reduce under the comm rule to give us

$$\bar{x}y.0 \mid x(z).P \;\rightarrow\; 0 \mid P\{y/z\}$$

and we have that $0 \mid P\{y/z\} \equiv P\{y/z\}$. By the way the comm rule interacts with parallel composition, we get

$$\bar{x}y.0 \mid x(z).P \mid Q \;\rightarrow\; P\{y/z\} \mid Q$$

(ii) $\bar{x}y.0 \mid x(u).P \mid x(v).Q$ does not reduce uniquely
Proof: We have two valid reductions as there are two possible comms that can occur. One reduction results from a comm between the first and second terms, i.e. reduces to

$$P\{y/u\} \mid x(v).Q,$$

and the other results from a comm between the first and third terms, i.e. reduces to

$$x(u).P \mid Q\{y/v\}$$

(iii) $(\nu x)\,(\bar{x}y.0 \mid x(u).P) \mid x(v).Q \;\rightarrow\; P\{y/u\} \mid x(v).Q$
Proof: The restriction of the name x to the first two terms in the parallel composition results in a unique reduction in this case, as opposed to the previous example. Here the name x restricted by (νx) is distinct from the name x appearing in the term $x(v).Q$. Therefore, only the first two terms comm resulting in the reduction

$$P\{y/u\} \mid x(v).Q$$

(iv) $!\bar{x}y.0 \mid x(u).P \;\rightarrow !\bar{x}y.0 \mid P\{y/u\}$
Proof: We know that $!\bar{x}y.0 \equiv !\bar{x}y.0 \mid \bar{x}y.0$ because replication always allows for the creation of another copy. Now we have a comm between $\bar{x}y.0$ and $x(u).P$, which we have seen reduces to $P\{y/u\}$. Thus, we are left with

$$!\bar{x}y.0 \mid P\{y/u\}$$

To summarize, the π-calculus is a concurrent model of computation based on the notion of naming, in which we represent interactions between agents as message-passing. This message-passing occurs when we have concurrent send and receive processes on the same channel. In π-calculus, we make a distinction between names and processes and we have the option to run computations concurrently. These are the major differences with λ-calculus.

To see the full version of π-calculus, see [1]. We have not discussed the choice operator and normal processes because our intention in introducing π-calculus was to investigate concurrency and how that is dealt with in a grammar.

The goals of this computational calculi primer series are to introduce ρ-calculus and set the foundation for the DoCC lectures. We will achieve both of these goals with next the article.

1.4 References

1. Robin Milner, *The Polyadic π-Calculus: a Tutorial*