Supplementary Notes on The Pi-Calculus

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 24
November 19, 2002

In this lecture we first consider the question of observational equivalence for the calculus of concurrent, communicating processes. Then we extend the calculus to allow us communication to transmit values, which leads to the π-calculus.

Recall from the last lecture our notion of process expression, and in particular the unobservable (internal) action τ.

$$\begin{array}{rrcl} \text{Observable Action} & \lambda & ::= & \mathrm{a} \mid \overline{\mathrm{a}} \\ \text{Action} & \alpha & ::= & \lambda \mid \tau \\ \text{Process Exps} & P & ::= & A\langle a_1, \ldots, a_n\rangle \mid N \mid (P_1 \mid P_2) \mid \mathsf{new}\, a.P \\ \text{Sums} & N & ::= & \alpha.P \mid N_1 + N_2 \mid 0 \end{array}$$

The operational semantics with observable behavior is given by the judgment $P \xrightarrow{\alpha} P'$ which is defined by the following rules. Here we write $\overline{\lambda}$ for the opposite of λ, with the understanding that $\overline{\overline{\mathrm{a}}} = a$.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum}_t \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{React}_t$$

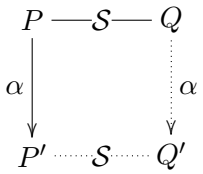
$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{L-Par}_t \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\ \text{R-Par}_t$$

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin \{\mathrm{a}, \overline{\mathrm{a}}\})}{\mathsf{new}\, a.P \xrightarrow{\alpha} \mathsf{new}\, a.P'}\ \text{Res}_t$$

$$\frac{\{b_1/a_1, \ldots, b_n/a_n\}P_A \xrightarrow{\alpha} P' \quad (A(a_1, \ldots, a_n) \stackrel{\text{def}}{=} P_A)}{A\langle b_1, \ldots, b_n\rangle \xrightarrow{\alpha} P'}\ \text{Ident}_t$$

Also recall our definition of a *strong simulation* $\mathcal{S}$: If $P \mathcal{S} Q$ and $P \xrightarrow{\alpha} P'$ then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$.

In pictures:



where the solid lines indicate given relationships and the dotted lines indicate the relationships whose existence we have to verify (including the existence of Q'). If such a strong simulation exists, we say that Q strongly simulates P.

Futhermore, we say that two states are *strongly bisimilar* if there is a single relation $\mathcal{S}$ such that both the relation and its converse are strong simulations.

Strong simulation does not distinguish between silent (also called internal or unobservable) transitions τ and observable transitions λ (consisting either of names a or co-names $\overline{\mathrm{a}}$). When considering the observable behavior of a process we would like to "ignore" silent transitions to some extent. Of course, this is not entirely possibly, since a silent transition can change from a state with many enabled actions to one with much fewer or different ones. However, we can allow any number of internal actions in order to simulate a transition. We define

$$\begin{array}{lll} P \xrightarrow{\tau^*} P' & \text{iff} & P \xrightarrow{\tau} \cdots \xrightarrow{\tau} P' \\ P \xrightarrow{\tau^* \lambda \tau^*} P' & \text{iff} & P \xrightarrow{\tau^*} P_1 \xrightarrow{\lambda} P_2 \xrightarrow{\tau^*} P' \end{array}$$

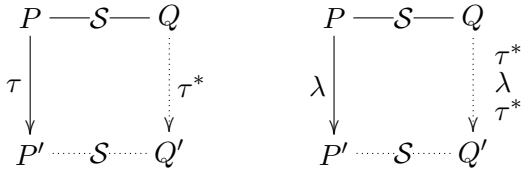
In particular, we always have $P \xrightarrow{\tau^*} P$. Then we say that $\mathcal{S}$ is a *weak simulation* if the following two conditions are satisfied:[1]

(i) If $P \mathcal{S} Q$ and $P \xrightarrow{\tau} P'$
then there exists a Q' such that $Q \xrightarrow{\tau^*} Q'$ and $P' \mathcal{S} Q'$.

(ii) If $P \mathcal{S} Q$ and $P \xrightarrow{\lambda} P'$
then there exists a Q' such that $Q \xrightarrow{\tau^* \lambda \tau^*} Q'$ and $P' \mathcal{S} Q'$.

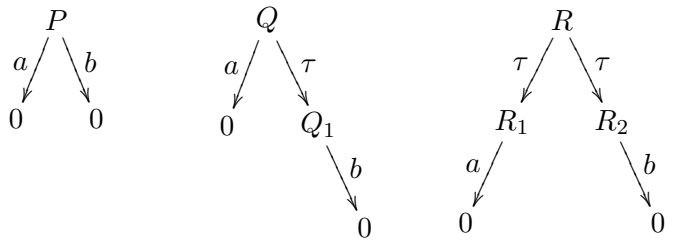
[1] This differs slightly, but I believe insignificantly from Milner's definition.

In pictures:



As before we say that Q *weakly simulates* P if there is a weak simulation $\mathcal{S}$ with $P \; \mathcal{S} \; Q$. We say P and Q are *weakly bisimilar* if there is a relation $\mathcal{S}$ such that both $\mathcal{S}$ and its inverse are weak simulations. We write $P \approx Q$ if P and Q are weakly bisimular.

We can see that the relation of weak bisimulation concentrates on the externally observable behavior. We show some examples that demonstrate processes that are *not* weakly bisimilar.



$$P = a.0 + b.0 \qquad Q = a.0 + \tau.b.0 \qquad R = \tau.a.0 + \tau.b.0$$

Even though P, Q, and R can all weakly simulate each other, no two are weakly bisimilar. As an example, consider P and Q. Then any weak bisimulation must relate P and Q_1, because if $Q \xrightarrow{\tau} Q_1$ then P can match this only by idling (no transition). But $P \xrightarrow{a} 0$ and Q_1 cannot match this step. Therefore P and and Q cannot be weakly bisimilar. Analogous arguments suffice for the other pairs of processes.

As positive examples of weak bisimulation, we have

$$\begin{aligned} a.P &\approx \tau.a.P \\ a.P + \tau.a.P &\approx \tau.a.P \\ a.(b.P + \tau.c.Q) &\approx a.(b.P + \tau.c.Q) + \tau.c.Q \end{aligned}$$

The reader is encouraged to draw the corresponding transition diagrams. As an example, consider the second equation.

$$Q_1 = a.P + \tau.a.P \quad \text{and} \quad Q_2 = \tau.a.P$$

We relate $Q_1 \; \mathcal{S} \; Q_2$ and $a.P \; \mathcal{S} \; a.P$ and $P \; \mathcal{S} \; P$. In one direction we have

1. $Q_1 \xrightarrow{a} P$ which can be simulated by $Q_2 \xrightarrow{\tau\, a} P$.
2. $Q_1 \xrightarrow{\tau} a.P$ which can be simulated by $Q_2 \xrightarrow{\tau} a.P$.

In the other direction we have

1. $Q_2 \xrightarrow{\tau} a.P$ which can be simulated by $Q_1 \xrightarrow{\tau} a.P$.

Together these cases yield the desirect result: $Q_1 \approx Q_2$.

Next we will generalize the calculus of concurrent processes so that value can be transmitted during communication. But our language has no primitive values, so this just reduces to transmitting names along channels that are themselves represented as names. This means that a system of processes can dynamically change its communication structure because connections to processes can be passed as first class values. This is why the resulting language, the π-calculus, is called a calculus of *mobile* and *concurrent* communicating processes.

We generalize actions and differentiate them more explicitly into input actions and output actions, since one side of a synchronized communication act has to send and the other to receive a name. We also replace primitive process identifiers and defining equation by process replication $!P$ explained below.

Action prefixes	π	$::=$	$\mathrm{x}(y)$	receive y along x
		$\mid$	$\overline{\mathrm{x}}\langle y\rangle$	send y along x
		$\mid$	τ	unobservable action
Process exps	P	$::=$	$N \mid (P_1 \mid P_2) \mid \mathsf{new}\, a.P \mid\, !P$	
Sums	N	$::=$	$0 \mid N_1 + N_2 \mid \pi.P$	

The structural congruence remains the same as before, except that in addition we have $!P \equiv P \mid\, !P$, that is, a process $!P$ can spawn arbitrarily many copies of itself.

In examples $\pi.0$ is often abbreviated by π. Note that in a summand $\mathrm{x}(y).P$, y is a *bound variable* with scope P that stands for the value received along x. On the other hand, $\overline{\mathrm{x}}\langle y\rangle.P$ does not bind any variables.

Before presenting the transition semantics, we consider the following example.

$$P = ((\overline{\mathrm{x}}\langle y\rangle.0 + \mathrm{z}(w).\overline{\mathrm{w}}\langle y\rangle.0) \mid \mathrm{x}(u).\overline{\mathrm{u}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0)$$

The middle process can synchronize and communicate with either the first or the last one. Reaction with the first leads to

$$P_1 = (0 \mid \overline{\mathrm{y}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0) \equiv (\overline{\mathrm{y}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0)$$

which cannot transition further. Reaction with the seconds leads to

$$P_1' = ((\overline{\mathrm{x}}\langle y\rangle.0 + \mathrm{z}(w).\overline{\mathrm{w}}\langle y\rangle.0) \mid \overline{\mathrm{z}}\langle v\rangle.0 \mid 0)$$

which can step further to

$$P_2' = (\overline{\mathrm{v}}\langle y\rangle.0 \mid 0 \mid 0)$$

Next we show the reaction rules in a form which does not make an externally observable action explicit, and exploits structural congruence.

$$\frac{}{\tau.P + N \longrightarrow P}\ \text{Tau}$$

$$\frac{}{(\mathrm{a}(x).P + M) \mid (\overline{\mathrm{a}}\langle b\rangle.Q + N) \longrightarrow (\{b/x\}P) \mid Q}\ \text{React}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{Par} \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, x.P \longrightarrow \mathsf{new}\, x.P'}\ \text{Res}$$

$$\frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}\ \text{Struct}$$

Even though the syntax does not formally distinguish, we use x for binding occurrences of names (subject to silent renaming), and a and b for non-binding occurrences.

As a simple example we will model a storage cell that can hold a value and service get and put requests to read and write the cell contents. We first show it using definitions for process identifiers and then rewrite it using process replication.

$$\begin{aligned} C(x, \mathrm{get}, \mathrm{put}) \quad \stackrel{\mathrm{def}}{=} \quad & \overline{\mathrm{get}}\langle x\rangle.C\langle x, \mathrm{get}, \mathrm{put}\rangle.0 \\ & + \mathrm{put}(y).C\langle y, \mathrm{get}, \mathrm{put}\rangle.0 \end{aligned}$$

We express this in the π-calculus by turning C itself into a name, left-hand side into an input action and occurrences on the right-hand side into an output action.

$$!\,\mathrm{c}(x, \mathrm{get}, \mathrm{put}).(\overline{\mathrm{get}}\langle x\rangle.\overline{\mathrm{c}}\langle x, \mathrm{get}, \mathrm{put}\rangle.0 + \mathrm{put}(y).\overline{\mathrm{c}}\langle y, \mathrm{get}, \mathrm{put}\rangle.0)$$

We abbreviate this process expression by $!C$. In order to be in the calculus we must be able to receive and send multiple names at once. It is

straightforward to add this capability. As an example, consider how to create cell with initial contents 3, write 4 to it, read the cell and then print the contents some output device. Printing a is represented by an output action $\overline{\mathrm{print}}\langle a\rangle.0$. We also consider 3 and 4 just as names here.

$$!C \mid \mathsf{new}\, g.\mathsf{new}\, p.\overline{\mathrm{c}}\langle 3, g, p\rangle.\overline{\mathrm{p}}\langle 4\rangle.\mathrm{g}(x).\overline{\mathrm{print}}\langle x\rangle.0$$

Note that c and print are the only free names in this expression. Note also that we are creating new names g and p to stand for the channel to get or put a names into the storage cell C. We leave it to the reader as an instructive exercise to simulate the behavior of this expression. It should be clear, however, that we need to use structural equivalence initially to obtain a copy of C with which we can react after moving the quantifiers of g and p outside.

As a more involved example, consider the following specification of the sieve of Eratosthenes. We start with a stream to produce integers, assuming we have a primitive successor operation on integer names.[2] The idea is to have a channel which sends successive numbers.

$$!\mathrm{count}(n, \mathrm{out}).\overline{\mathrm{out}}\langle n\rangle.\overline{\mathrm{count}}(n+1, \mathrm{out})$$

Second we show a process to filter all multiples of a given prime number from its input stream while producing the output stream. We assume an oracle $(\mathrm{x} \bmod \mathrm{p} = 0)$ and its negation.

$$\begin{aligned}!\mathrm{filter}(p, \mathrm{in}, \mathrm{out}).\mathrm{in}(x).((\mathrm{x} \bmod \mathrm{p} = 0)().\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{out}\rangle.0 \\ + (\mathrm{x} \bmod \mathrm{p} \neq 0)().\overline{\mathrm{out}}(x).\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{out}\rangle.0)\end{aligned}$$

Finally, we come to the process that generates a sequence of prime numbers, starting from the first item of the input channel which should be prime (by invariant).

$$\begin{aligned}!\mathrm{primes}(\mathrm{in}, \mathrm{out}).\mathrm{in}(p).\overline{\mathrm{out}}\langle p\rangle. \\ \mathsf{new}\, \mathrm{mid}.(\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \mid \overline{\mathrm{primes}}\langle \mathrm{mid}, \mathrm{out}\rangle.0)\end{aligned}$$

primes establishes a new filtering process for each prime and threads the input stream in into the filter. The first element of the filtered result stream is guaranteed to be prime, so we can invoke the primes process recursively.

At the top level, we start the process with the stream of numbers counting up from 2, the smallest prime. This will generate communication requests $\overline{\mathrm{out}}\langle p\rangle$ for each successive prime.

[2] This can also be coded in the π-calculus, but we prefer to avoid this complication here.

$$\mathsf{new}\,\mathrm{nats}.\overline{\mathrm{count}}\langle 2, \mathrm{nats}\rangle \mid \overline{\mathrm{primes}}\langle \mathrm{nats}, \mathrm{out}\rangle$$

In this implementation, communication is fully synchronous, that is, both sender and receiver can only move on once the message has been exchanged. Here, this means that the prime numbers are guaranteed to be read in their natural order. If we don't care about the order, we can rewrite the process so that it generates the primes *asynchronously*. For this we use the general transformation of

$$\overline{\mathrm{a}}\langle b\rangle.P \quad \Longrightarrow \quad \tau.(\overline{\mathrm{a}}\langle b\rangle.0 \mid P)$$

which means the computation of P can proceed regardless whether the message b has been received along channel a. In our case, this would be a simple change in the primes generator.

$$\begin{array}{l}!\mathrm{primes}(\mathrm{in}, \mathrm{out}).\mathrm{in}(p). \\ \quad \overline{\mathrm{out}}\langle p\rangle.0 \mid \mathsf{new}\,\mathrm{mid}.(\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \;\mid\; \overline{\mathrm{primes}}\langle \mathrm{mid}, \mathrm{out}\rangle.0)\end{array}$$

The advantage of an asynchronous calculus is its proximity to a realistic model of computation. On the other hand, synchronous communciation allows for significantly shorter code, because no protocol is neede to make sure messages have been received, and in received in order. Since asynchronous communication is very easily coded here, we stick to Milner's original π-calculus which was synchronous.

In the next lecture we will see how a variant of the π-calculus can be embedded in a full-scale language such as Standard ML to offer rich concurrency primitives in addition to functional programming.