Lecture Notes on Concurrent Processes

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 25
December 2, 2004

We have seen in the last lecture that by investigating the reactive behavior of systems, we obtain a very different view of computation. Instead of termination and the values of expressions, it is the interactions with the outside world that are of interest. As an example, we showed an important notion of program equivalence, namely strong bisimulation and contrasted it with observational equivalence of computation with respect to values.

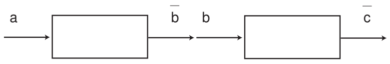
The processes we have considered so far were non-deterministic, but sequential. In this lecture we generalize this to allow for concurrency and also name restriction to obtain a form of abstraction.

In order to model concurrency we allow *process composition*, $P_1 \mid P_2$. Intuitively, this means that processes P_1 and P_2 execute concurrently. Such concurrent processes can interact in a synchronous fashion when one process wants to perform an input action and another process wants to perform a matching output action. As a very simple example, consider two processes A and B plugged together in the following way. A performs input action a and then wants to perform output action $\overline{b}$, returning to state A. Process B performs an input action b followed by an output action $\overline{c}$, returning to state B upon completion.

$$\begin{array}{lcl} A & \stackrel{\text{def}}{=} & a.\overline{b}.A \\ B & \stackrel{\text{def}}{=} & b.\overline{c}.B \end{array}$$

We can think of A of a transducer that transforms the input a to the output b, and similarly for B.

We assume we start with A and B operating concurrently, that is, plugged together, communicating along channel b



In process notation we just separate the processes by a bar "|"

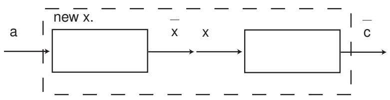
$$A \mid B$$

Now we can have the following sequence of transitions:

$$A \mid B \xrightarrow{a} \overline{b}.A \mid b.\overline{c}.B \longrightarrow A \mid \overline{c}.B \xrightarrow{\overline{c}} A \mid B$$

We have explicitly unfolded B after the first step to make the interaction between $\overline{b}$ and b clear. Note that this synchronization is not an external event, so the transition arrow is unadorned. We call this an *internal action* or *silent action* are write τ.

The second generalization from the sequential processes is to permit name hiding (abstraction). In the example above, we plugged processes A and B together, intuitively connecting the output $\overline{b}$ from A with the input b from B. However, it is still possible to put another process in parallel with A and B that could interact with both of them using b. In order to prohibit such behavior, we can locally bind the name b. We write $\mathsf{new}\, x.P$ for a process with a locally bound name x. Names bound with $\mathsf{new}\, x.P$ are subject to α-conversion (renaming of bound variables) as usual. In a picture:



We might try to express this as

$$\mathsf{new}\, b.A \mid B.$$

However, we have created a new problem: the name b is bound in this expression, but the scope of b does not include the definitions of A and B. In order to avoid this scope violation we parameterize the process definitions by all names that they use, and apply uses of the process identifier with the appropriate local names. We can think of this as a special form of parameter passing or renaming.

$$\begin{array}{rcl} A(x,y) & \stackrel{\mathrm{def}}{=} & x.\overline{y}.A\langle x,y\rangle \\ B(y,z) & \stackrel{\mathrm{def}}{=} & y.\overline{z}.B\langle y,z\rangle \end{array}$$

The process expression can now hygienically refer to locally bound names.

$$\mathsf{new}\, x.A\langle a, x\rangle \mid B\langle x, c\rangle$$

But the definitions of A and B are really the same up to α-conversion, so we can rewrite this also as

$$\mathsf{new}\, x.A\langle a, x\rangle \mid A\langle x, c\rangle$$

This leads to the following language of *concurrent process expressions*.

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

We define the operational semantics of concurrent processes with the set of rules below. In this semantics an action is made explicit in a transition, but matching input/output actions become silent. We use λ to stand for either a or $\overline{a}$ and $\overline{\lambda}$ for $\overline{a}$ or a, respectively.

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

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

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

$$\frac{\{a_1/x_1, \ldots, a_n/x_n\}P_A \xrightarrow{\alpha} P' \quad (A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A)}{A\langle a_1, \ldots, a_n\rangle \xrightarrow{\alpha} P'}\ \text{Ident}_t$$

If we want to examine the interaction of a system with its environment we consider the environment as another *testing process* that is run concurrently with the system whose behavior we wish to examine. In this situation we are mostly interested in silent transitions, since the interactions of a process with its environment are explicit and therefore silent. As example for the above rules, consider the following process expression.

$$P = (\mathsf{new}\, a.((a.Q_1 + b.Q_2) \mid \overline{a}.0)) \mid (\overline{b}.R_1 + \overline{a}.R_2)$$

Note that the output action before R_2 is a different name than a used as the input action to Q_1, the latter being locally quantified. This means there are only two possible τ-transitions.

$$\begin{array}{l} P \longrightarrow (\mathsf{new}\, a.(Q_1 \mid 0)) \mid (\overline{b}.R_1 + \overline{a}.R_2) \\ P \longrightarrow (\mathsf{new}\, a.(Q_2 \mid \overline{a}.0)) \mid R_1 \end{array}$$

As another example[1] of this form of concurrent processes, consider two two-way transducers of identical structure.

$$A(a, a', b, b') \quad \stackrel{\mathrm{def}}{=} \quad a.\overline{b}.A\langle a, a', b, b'\rangle + b'.\overline{a'}.A\langle a, a', b, b'\rangle$$

We now compose to instances of this process concurrently, hiding the internal connection between.

$$\mathsf{new}\, b.\mathsf{new}\, b'.(A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle)$$

At first one might suspect this is bisimilar with $A\langle a, a', c, c'\rangle$, which shortcircuits the internal synchronization along b and b'. While we have not formally defined bisimilarity in this new setting, this new composition is in fact buggy: it can deadlock when put in parallel with $\overline{a}.P$, $c.P'$, $\overline{c'}.Q$, $a'.Q'$

$$\begin{array}{l} \overline{a}.P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle) \\ \longrightarrow P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{b}.A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle) \\ \longrightarrow P \mid c.P' \mid Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{b}.A\langle a, a', b, b'\rangle \mid \overline{b'}.A\langle b, b', c, c'\rangle) \end{array}$$

At this point all interactions are blocked and we have a deadlock. This can not happen with the process $A\langle a, a', c, c'\rangle$. It can evolve in different ways but not deadlock in the manner above; here is an example.

$$\begin{array}{l} \overline{a}.P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid A\langle a, a', c, c'\rangle \\ \longrightarrow P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \overline{c}.A\langle a, a', c, c'\rangle \\ \longrightarrow P \mid P' \mid \overline{c'}.Q \mid a'.Q' \mid A\langle a, a', c, c'\rangle \\ \longrightarrow P \mid P' \mid Q \mid a'.Q' \mid \overline{a'}.A\langle a, a', c, c'\rangle \\ \longrightarrow P \mid P' \mid Q \mid Q' \mid A\langle a, a', c, c'\rangle \end{array}$$

The reader should make sure to understand these transition and redesign the composed two-way buffer so that this deadlock situation cannot occur.

[1]not discussed in lecture

Observational Equivalence for Concurrent Processes .

Next we consider the question of observational equivalence for the calculus of concurrent, communicating processes.

Recall from the last lecture our definition of a *strong simulation* $\mathcal{S}$: If $P \mathrel{\mathcal{S}} Q$ and $P \xrightarrow{\alpha} P'$ then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathrel{\mathcal{S}} Q'$.

In pictures:

$$\begin{tikzcd}
P \arrow[r, dash, "\mathcal{S}"] \arrow[d, "\alpha"'] & Q \arrow[d, dotted, "\alpha"] \\
P' \arrow[r, dotted, dash, "\mathcal{S}"] & Q'
\end{tikzcd}$$

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 $\overline{a}$). 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:[2]

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

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

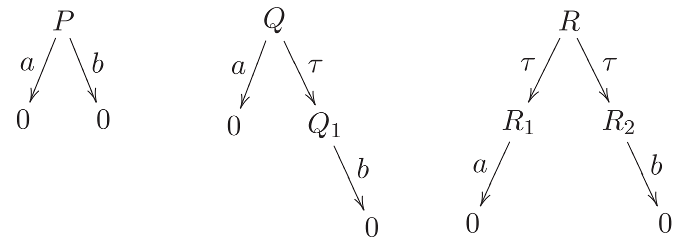
[2] This differs slightly, but I believe insignificantly from Milner's definition.

In pictures:

$$\begin{tikzcd} P \arrow[r, dash, "\mathcal{S}"] \arrow[d, "\tau"'] & Q \arrow[d, dotted, "\tau^*"] \\ P' \arrow[r, dotted, dash, "\mathcal{S}"] & Q' \end{tikzcd} \qquad \begin{tikzcd} P \arrow[r, dash, "\mathcal{S}"] \arrow[d, "\lambda"'] & Q \arrow[d, dotted, "\tau^* \lambda \tau^*"] \\ P' \arrow[r, dotted, dash, "\mathcal{S}"] & Q' \end{tikzcd}$$

As before we say that Q *weakly simulates* P if there is a weak simulation $\mathcal{S}$ with $P \mathrel{\mathcal{S}} Q$. 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

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

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 $Q_1 \mathrel{\mathcal{S}} Q_2$ and $a.P \mathrel{\mathcal{S}} a.P$ and $P \mathrel{\mathcal{S}} P$. 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 desired result: $Q_1 \approx Q_2$.

As a final example, we return to the earlier simple transducer.

$$A(x, y) \stackrel{\text{def}}{=} x.\overline{y}.A\langle x, y\rangle$$

Let us compare the two transducers $A\langle a, c\rangle$ and $\mathsf{new}\, x.\ A\langle a, x\rangle \mid A\langle x, c\rangle$. We can easily see that they are not strongly bisimilar, because the latter process can make a silent transition that the first one can not.

But are they weakly bisimilar? It seems both processes can only input a's and produce the same number of c's. Since the intermediate channel x is locally quantified, no other communication with the composed process can take place.

And yet, the two are not weakly bisimilar. To see this, consider the testing process $\overline{a}.\overline{a}.0$. If we put this in parallel with the first process, we obtain

$$\overline{a}.\overline{a}.0 \mid A(a, c) \xrightarrow{\tau} \overline{a}.0 \mid \overline{c}.A\langle a, c\rangle$$

However, if we put it in parallel with the second, we can make further process:

$$\begin{array}{l}
\overline{a}.\overline{a}.0 \mid \mathsf{new}\, x.\ A\langle a, x\rangle \mid A\langle x, c\rangle \\
\xrightarrow{\tau} \overline{a}.0 \mid \overline{x}.A\langle a, x\rangle \mid A\langle x, c\rangle \\
\xrightarrow{\tau} \overline{a}.0 \mid A\langle a, x\rangle \mid \overline{c}.A\langle x, c\rangle \\
\xrightarrow{\tau} 0 \mid \overline{x}.A\langle a, x\rangle \mid \overline{c}.A\langle x, c\rangle \\
\approx \overline{x}.A\langle a, x\rangle \mid \overline{c}.A\langle x, c\rangle
\end{array}$$

More formally, we can prove that they are not bisimilar by assuming that they are and then deriving a contradiction. The contradication can be seen by constructing the following diagram. We construct this by choosing transitions on the left and simulating them on the write. We use τ_0 to stand for idling (= no transition) which is allowed under weak bisimulation.

$$
\begin{tikzcd}
\mathsf{new}\, x.\, A\langle a, x\rangle \mid A\langle x, c\rangle \arrow[d, "a"'] & \approx & A\langle a, c\rangle \arrow[d, "a"] \\
\mathsf{new}\, x.\, \overline{x}.A\langle a, x\rangle \mid A\langle x, c\rangle \arrow[d, "\tau"'] & \approx & \overline{c}.A\langle a, c\rangle \arrow[d, "\tau^0"] \\
\mathsf{new}\, x.\, A\langle a, x\rangle \mid \overline{c}.A\langle x, c\rangle \arrow[d, "a"'] & \approx & \overline{c}.A\langle a, c\rangle \arrow[d, dotted, "a"] \\
\mathsf{new}\, x.\, \overline{x}.A\langle a, x\rangle \mid \overline{c}.A\langle x, c\rangle & \approx & ?
\end{tikzcd}
$$

The last row gives the desired contradiction, since the process $\overline{c}.A\langle a, c\rangle$ cannot exhibit an a transition. Essentially, the transducer on the left has an internal buffer of since one, while the one on the right does not. This shows that in many practical examples, we may need yet other notions of equivalence.

In the next lecture we extend extend the calculus to allow us communication to transmit values, which leads to the π-calculus. Then 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.