

Concepts of Concurrent Computation

Spring 2015

Lecture 10: CCS

Sebastian Nanz
Chris Poskitt

Process calculi

- Question: Why do we need a theoretical model of concurrent computation?
- Turing machines or the λ -calculus have proved to be useful models of sequential systems
- Abstracting away from implementation details yields general insights into programming and computation
- Process calculi help to focus on the essence of concurrent systems: [interaction](#)

The Calculus of Communicating Systems

- We study the **Calculus of Communicating Systems (CCS)** [Milner 1980]
- Milner's general model:
 - A **concurrent system** is a collection of processes
 - A **process** is an independent agent that may perform internal activities in isolation or may interact with the environment to perform shared activities
- Milner's insight: Concurrent processes have an algebraic structure

$$\boxed{P1} \text{ op } \boxed{P2} \Rightarrow \boxed{P1 \text{ op } P2}$$

- This is why a process calculus is sometime called a **process algebra**

Example: A simple process

- A coffee and tea machine may take an order for either tea or coffee, accept the appropriate payment, pour the ordered drink, and terminate:

$$tea.coin.\overline{cup_of_tea}.0 + coffee.coin.coin.\overline{cup_of_coffee}.0$$

- We have the following elements of syntax:
 - **Actions:** tea , $\overline{cup_of_tea}$, etc.
 - **Sequential composition:** the dot “.” (first do action tea , then $coin$, ...)
 - **Non-deterministic choice:** the plus “+” (either do tea or $coffee$)
 - **Terminated process:** 0

Example: Execution of a simple process

- When a process executes it performs some action, and becomes a new process
- The execution of an action a is symbolized by a transition \xrightarrow{a}

$$\begin{array}{r}
 \overline{\text{tea.coin.cup_of_tea.0}} + \overline{\text{coffee.coin.coin.cup_of_coffee.0}} \\
 \xrightarrow{\text{tea}} \quad \overline{\text{coin.cup_of_tea.0}} \\
 \xrightarrow{\text{coin}} \quad \overline{\text{cup_of_tea.0}} \\
 \xrightarrow{\text{cup_of_tea}} \quad 0
 \end{array}$$

Syntax of CCS

Syntax of CCS

- Goal: In the following we introduce the syntax of CCS step-by-step
- Basic principle
 1. Define **atomic processes** that model the simplest possible behavior
 2. Define **composition operators** that build more complex behavior from simpler ones

The terminal process

- The simplest possible behavior is **no behavior**
- We write **0** (pronounced “nil”) for the **terminal** or **inactive process**
 - **0** models a system that is either deadlocked or has terminated
 - **0** is the only atomic process of CCS

Names and actions

- We assume an infinite set \mathcal{A} of **port names**, and a set $\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$ of **complementary port names**
- Input actions
 - When modeling we use a name a to denote an **input action**, i.e. the receiving of input from the associated port a
- Output actions
 - We use a co-name \bar{a} to denote an **output action**, i.e. the sending of output to the associated port a
- Internal actions
 - We use τ to denote the distinguished **internal action**
- The set of actions Act is given by $Act = \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$

Action prefixing

- The simplest actual behavior is sequential behavior
- Action prefixing
 - If P is a process we write

$$\alpha.P$$

to denote the prefixing of P with the action α

- $\alpha.P$ models a system that is ready to perform the action, α , and then behaves as P , i.e.

$$\alpha.P \xrightarrow{\alpha} P$$

Example: Action prefixing

- A process that starts a timer, performs some internal computation, and then stops the timer:

$$\overline{go}.\tau.\overline{stop}.0 \xrightarrow{\overline{go}} \tau.\overline{stop}.0 \xrightarrow{\tau} \overline{stop}.0 \xrightarrow{\overline{stop}} 0$$

Process interfaces

- Interfaces
 - The set of **input and output actions** that a process P may perform in isolation constitutes the **interface** of P
 - The interface enumerates the ports that P may use to interact with the environment
- Example: The interface of the coffee and tea machine is

tea, coffee, coin, $\overline{cup_of_tea}$, $\overline{cup_of_coffee}$

Non-deterministic choice

- A more advanced sequential behavior is that of **alternative behaviors**
- **Non-deterministic choice**
 - If P and Q are processes then we write

$$P + Q$$

to denote the **non-deterministic choice** between P and Q

- $P + Q$ models a process that can either behave as P (discarding Q) or as Q (discarding P)

Example: Non-deterministic choice

$$\overline{tea.coin.cup_of_tea}.0 + \overline{coffee.coin.coin.cup_of_coffee}.0 \xrightarrow{tea} \overline{coin.cup_of_tea}.$$

- Note
 - Prefixing binds harder than plus
 - The choice is made by the initial *coffee / tea* button press

Process constants and recursion

- The most advanced sequential behavior is **recursive behavior**
- **Process constants**
 - A process may be the invocation of a **process constant**, $K \in \mathcal{K}$
 - This is only meaningful if K is defined beforehand
- **Recursive definition**
 - If K is a process constant and P is a process we write

$$K \stackrel{\text{def}}{=} P$$

to give a **recursive definition** of the behavior of K
(recursive if P invokes K)

Example: Recursion (1)

- A system clock, SC , sends out regular clock signals forever:

$$SC \stackrel{\text{def}}{=} \overline{tick}.SC$$

- The system SC may behave as:

$$\overline{tick}.SC \xrightarrow{\overline{tick}} SC \xrightarrow{\overline{tick}} \dots$$

Example: Recursion (2)

- A fully automatic coffee and tea machine CTM

$$\text{CTM} \stackrel{\text{def}}{=} \overline{\text{tea.coin.cup_of_tea.CTM}} + \overline{\text{coffee.coin.coin.cup_of_coffee.CTM}}$$

- The system CTM may e.g. do:

$$\overline{\text{tea.coin.cup_of_tea.CTM}} + \overline{\text{coffee.coin.coin.cup_of_coffee.CTM}}$$

$$\begin{array}{l} \xrightarrow{\text{tea}} \quad \overline{\text{coin.cup_of_tea.CTM}} \\ \xrightarrow{\text{coin}} \quad \overline{\text{cup_of_tea.CTM}} \\ \xrightarrow{\text{cup_of_tea}} \quad \text{CTM} \\ \xrightarrow{\alpha} \quad \dots \end{array}$$

- This will serve drinks ad infinitum

Parallel composition

- Finally: concurrent behavior
- Parallel composition
 - If P and Q are processes we write

$$P \mid Q$$

to denote the **parallel composition** of P and Q

- $P \mid Q$ models a process that behaves like P and Q in parallel:
 - Each may proceed independently
 - If P is ready to perform an action a and Q is ready to perform the complementary action \bar{a} , they may **interact**

Example: Parallel composition

- Recall the coffee and tea machine:

$$CTM \stackrel{\text{def}}{=} \overline{tea.coin.cup_of_tea}.CTM + \overline{coffee.coin.coin.cup_of_coffee}.CTM$$

- Now consider a regular customer, the Computer Scientist CS:

$$CS \stackrel{\text{def}}{=} \overline{tea.coin.cup_of_tea}.teach.CS + \overline{coffee.coin.coin.cup_of_coffee}.publish.CS$$

- CS must drink coffee to publish
- CS can only teach on tea

Example: Parallel composition

- On an average Tuesday morning the system

$$\text{CTM} \mid \text{CS}$$

is likely to behave as follows:

$$\begin{array}{l}
 (\overline{\text{tea}}.\text{coin}.\overline{\text{cup_of_tea}}.\text{CTM} + \overline{\text{coffee}}.\text{coin}.\text{coin}.\overline{\text{cup_of_coffee}}.\text{CTM}) \\
 \mid \\
 (\overline{\text{tea}}.\overline{\text{coin}}.\overline{\text{cup_of_tea}}.\overline{\text{teach}}.\text{CS} + \overline{\text{coffee}}.\overline{\text{coin}}.\overline{\text{coin}}.\overline{\text{cup_of_coffee}}.\overline{\text{publish}}.\text{CS}) \\
 \xrightarrow{\tau} (\overline{\text{coin}}.\overline{\text{cup_of_tea}}.\text{CTM}) \mid (\overline{\text{coin}}.\overline{\text{cup_of_tea}}.\overline{\text{teach}}.\text{CS}) \\
 \xrightarrow{\tau} (\overline{\text{cup_of_tea}}.\text{CTM}) \mid (\overline{\text{cup_of_tea}}.\overline{\text{teach}}.\text{CS}) \\
 \xrightarrow{\tau} \text{CTM} \mid (\overline{\text{teach}}.\text{CS}) \\
 \xrightarrow{\overline{\text{teach}}} \text{CTM} \mid \text{CS}
 \end{array}$$

- Note that the synchronisation of actions such as $\text{tea} / \overline{\text{tea}}$ is expressed by a τ -action (i.e. regarded as an [internal step](#))

Restriction

- We control unwanted interactions with the environment by restricting the scope of port names
- Restriction
 - if P is a process and A is a set of port names we write

$$P \setminus A$$

for the restriction of the scope of each name in A to P

- Removes each name $a \in A$ and the corresponding co-name \bar{a} from the interface of P
- Makes each name $a \in A$ and the corresponding co-name \bar{a} inaccessible to the environment

Example: Restriction

- Recall the coffee and tea machine and the computer scientist:

$$\text{CTM} \mid \text{CS}$$

- Restricting the coffee and tea machine on *coffee* makes the coffee-button inaccessible to the computer scientist:

$$(\text{CTM} \setminus \{\text{coffee}\}) \mid \text{CS}$$

- As a consequence CS can only teach, and never publish

Summary: Syntax of CCS

$P ::= K$		process constants ($K \in \mathcal{K}$)
$\alpha.P$		prefixing ($\alpha \in Act$)
$\sum_{i \in I} P_i$		summation (I is an arbitrary index set)
$P_1 P_2$		parallel composition
$P \setminus L$		restriction ($L \subseteq \mathcal{A}$)

- The set of all terms generated by the abstract syntax is called **CCS process expressions**
- Notation

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i$$

$$Nil = 0 = \sum_{i \in \emptyset} P_i$$

CCS programs

- CCS program

- A collection of defining equations of the form

$$K \stackrel{\text{def}}{=} P$$

where $K \in \mathcal{K}$ is a process constant and $P \in \mathcal{P}$ is a CCS process expression

- Only one defining equation per process constant
- Recursion is allowed: e.g. $A \stackrel{\text{def}}{=} \bar{a}.A \mid A$
- The program itself gives only the definitions of process constants: we can only execute processes (which can however mention the process constants defined in the program)

Exercise: Syntax of CCS

- Which of the following expressions are correctly built CCS expressions?
 - Assume that A, B are process constants and that a, b are port names

$a.b.A + B$ ✓

$(a.0 + \bar{a}.A) \setminus \{a, b\}$ ✓

$(a.0 \mid \bar{a}.A) \setminus \{a, \tau\}$ ✗

$\tau.\tau.B + 0$ ✓

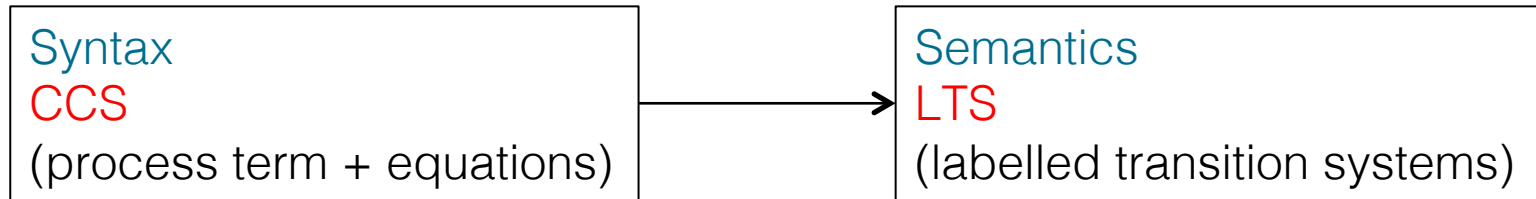
$(a.b.A + \bar{a}.0) \mid B$ ✓

$(a.b.A + \bar{a}.0).B$ ✗

Operational Semantics of CCS

Operational semantics

- Goal: Formalize the execution of a CCS process



Labelled transition systems

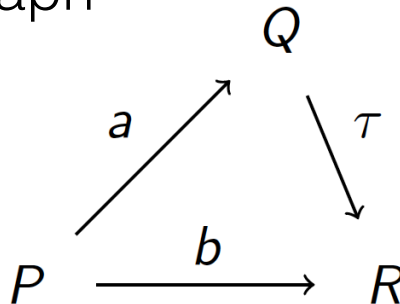
- A labelled transition system (LTS) is a triple $(Proc, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$ where
 - $Proc$ is a set of processes (the states),
 - Act is a set of actions (the labels), and
 - for every $\alpha \in Act$, $\xrightarrow{\alpha} \subseteq Proc \times Proc$ is a binary relation on processes called the transition relation
- We use the infix notation $P \xrightarrow{\alpha} P'$ to say that $(P, P') \in \xrightarrow{\alpha}$
- It is customary to distinguish the initial process (the start state)

Labelled transition systems

- Conceptually it is often beneficial to think of a (finite) LTS as something that can be drawn as a directed (process) graph
 - Processes are the nodes
 - Transitions are the edges
- Example: The LTS

$$\{\{P, Q, R\}, \{a, b, \tau\}, \{P \xrightarrow{a} Q, P \xrightarrow{b} R, Q \xrightarrow{\tau} R\}\}$$

corresponds to the graph



- Question: How can we produce an LTS (semantics) of a process term (syntax)?

Informal translation

- Terminal process: 0

behavior: $0 \not\rightarrow$

- Action prefixing: $\alpha.P$

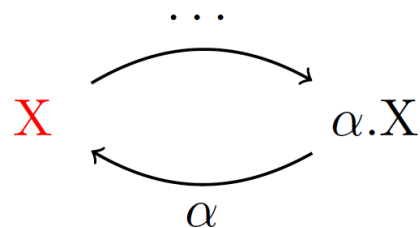
behavior: $\alpha.P \xrightarrow{\alpha} P$

- Non-deterministic choice: $\alpha.P + \beta.Q$

behavior: $P \xleftarrow{\alpha} \alpha.P + \beta.Q \xrightarrow{\beta} Q$

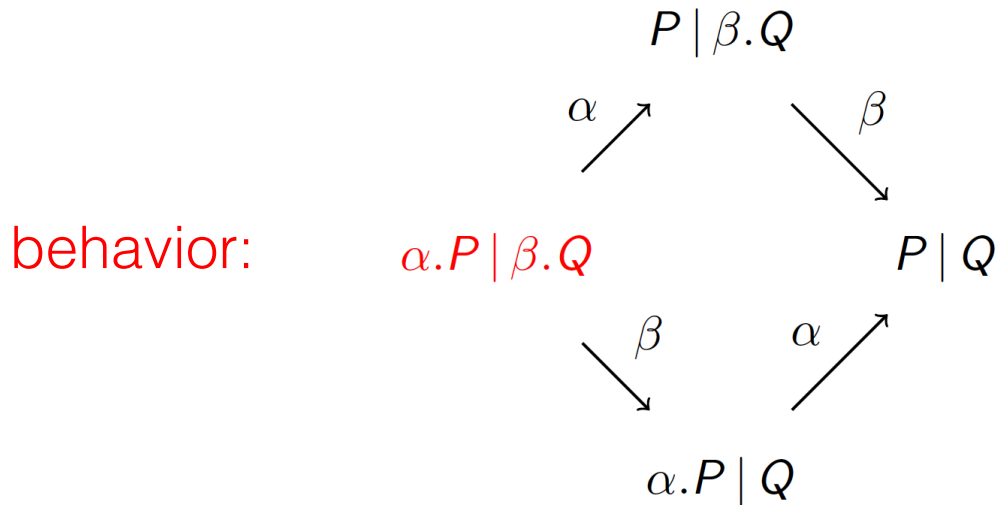
- Recursion: $X \stackrel{\text{def}}{=} \dots .\alpha.X$

behavior:



Informal translation

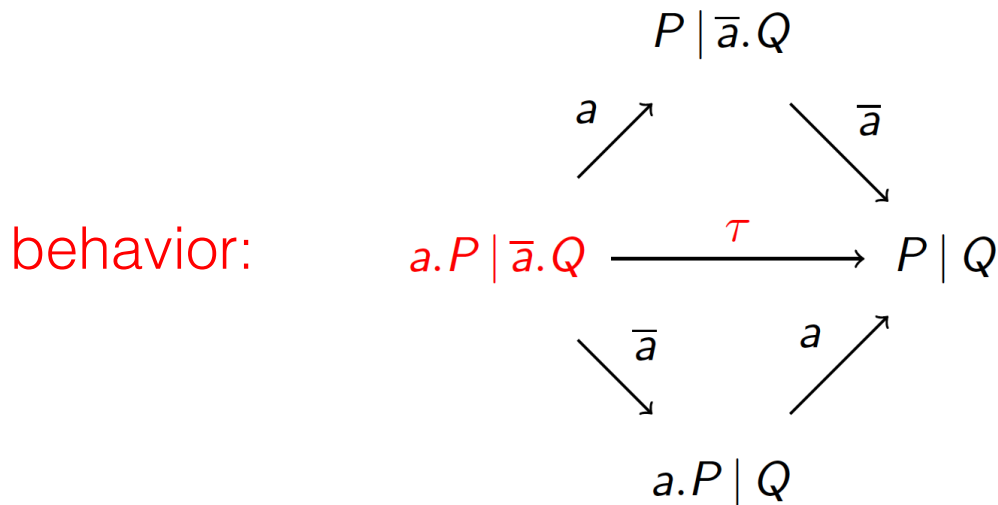
- Parallel composition: $\alpha.P \mid \beta.Q$
- Combines sequential composition and choice to obtain interleaving



- What about interaction?

Process interaction

- Concurrent processes, i.e. P and Q in $P \mid Q$, may interact where their interfaces are **compatible**
- A synchronizing interaction between two processes (subsystems), P and Q , is an activity that is **internal** to $P \mid Q$
- Parallel composition: $\alpha.P \mid \beta.Q$
- Allows **interaction** if $\beta = \bar{\alpha}$



Structural operational semantics for CCS

- Structural operational semantics (SOS) [Plotkin 1981]
 - Small-step operational semantics where the behavior of a system is inferred using syntax driven rules
- Given a collection of CCS defining equations, we define the LTS $(Proc, Act, \{\xrightarrow{a} \mid a \in Act\})$
 - $Proc$ is the set of all CCS process expressions
 - Act is the set of all CCS actions including
 - the transition relation is given by SOS rules of the form:

$$\text{RULE } \frac{\textit{premises}}{\textit{conclusion}} \textit{conditions}$$

SOS rules for CCS

$$\text{ACT} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM}_j \quad \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad j \in I$$

$$\text{COM1} \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{COM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\text{COM3} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\text{RES} \quad \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad \alpha, \bar{\alpha} \notin L \qquad \text{CON} \quad \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \quad K \stackrel{\text{def}}{=} P$$

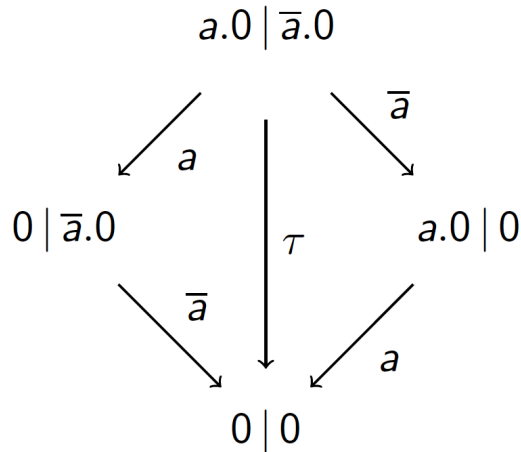
Example: Derivations

- Let $A \stackrel{\text{def}}{=} a.A$. Show that $((A | \bar{a}.0) | b.0) \xrightarrow{a} ((A | \bar{a}.0) | b.0)$

$$\begin{array}{c}
 \text{ACT} \frac{}{a.A \xrightarrow{a} A} \\
 \text{CON} \frac{a.A \xrightarrow{a} A}{A \xrightarrow{a} A} \quad A \stackrel{\text{def}}{=} a.A \\
 \text{COM1} \frac{}{A | \bar{a}.0 \xrightarrow{a} A | \bar{a}.0} \\
 \text{COM1} \frac{}{((A | \bar{a}.0) | b.0) \xrightarrow{a} ((A | \bar{a}.0) | b.0)}
 \end{array}$$

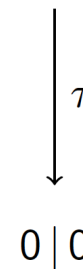
Restriction and interaction

- Restriction can be used to produce **closed systems**, i.e. their actions can only be taken internally (visible as τ -actions)



LTS of $a.0 \mid \bar{a}.0$

$$(a.0 \mid \bar{a}.0) \setminus \{a\}$$



LTS of $(a.0 \mid \bar{a}.0) \setminus \{a\}$

Conclusion

- Process calculi provide models of concurrency, not programming languages – for “everyday use” too many details are abstracted away
- However, the formal techniques studied in process calculi can help to design better concurrent programming languages