Concepts of Concurrent Computation

Spring 2015

Lecture 10: CCS

Sebastian Nanz

Chris Poskitt

Chair of Software Engineering

ETH zürich

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

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

- 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 $\xrightarrow{a}$

$$
\begin{array}{rl}
 & tea.coin.\overline{cup\_of\_tea}.0 + coffee.coin.coin.\overline{cup\_of\_coffee}.0 \\
\xrightarrow{tea} & coin.\overline{cup\_of\_tea}.0 \\
\xrightarrow{coin} & \overline{cup\_of\_tea}.0 \\
\xrightarrow{\overline{cup\_of\_tea}} & 0
\end{array}
$$

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 $\overline{a}$ 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

$$\mathit{tea}, \mathit{coffee}, \mathit{coin}, \overline{\mathit{cup\_of\_tea}}, \overline{\mathit{cup\_of\_coffee}}$$

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

$$\begin{aligned} & tea.coin.\overline{cup\_of\_tea}.0 \; + \; coffee.coin.coin.\overline{cup\_of\_coffee}.0 \\ \xrightarrow{tea} \quad & coin.\overline{cup\_of\_tea}. \end{aligned}$$

- 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, $\mathrm{K} \in \mathcal{K}$
  - This is only meaningful if K is defined beforehand
- Recursive definition
  - If K is a process constant and P is a process we write

$$\mathrm{K} \stackrel{\mathsf{def}}{=} P$$

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

Example: Recursion (1)

- A system clock, $\mathbf{SC}$, sends out regular clock signals forever:

$$\mathrm{SC} \stackrel{\mathsf{def}}{=} \overline{tick}.\mathrm{SC}$$

- The system $\mathbf{SC}$ may behave as:

$$\overline{tick}.\mathrm{SC} \xrightarrow{\overline{tick}} \mathrm{SC} \xrightarrow{\overline{tick}} \ldots$$

Example: Recursion (2)

- A fully automatic coffee and tea machine $\mathbf{CTM}$

$$\mathrm{CTM} \stackrel{\text{def}}{=} \mathit{tea}.\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM} + \mathit{coffee}.\mathit{coin}.\mathit{coin}.\overline{\mathit{cup\_of\_coffee}}.\mathrm{CTM}$$

- The system $\mathbf{CTM}$ may e.g. do:

$$\begin{aligned}
&\mathit{tea}.\mathit{coin}.\overline{\mathit{cup_of_tea}}.\mathrm{CTM} + \mathit{coffee}.\mathit{coin}.\mathit{coin}.\overline{\mathit{cup_of_coffee}}.\mathrm{CTM} \\
&\quad \xrightarrow{\mathit{tea}} \quad \mathit{coin}.\overline{\mathit{cup_of_tea}}.\mathrm{CTM} \\
&\quad \xrightarrow{\mathit{coin}} \quad \overline{\mathit{cup_of_tea}}.\mathrm{CTM} \\
&\quad \xrightarrow{\overline{\mathit{cup_of_tea}}} \quad \mathrm{CTM} \\
&\quad \xrightarrow{\alpha} \quad \ldots
\end{aligned}$$

- 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 $\overline{a}$, they may interact

Example: Parallel composition

- Recall the coffee and tea machine:

$$\mathrm{CTM} \stackrel{\mathsf{def}}{=} \mathit{tea}.\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM} + \mathit{coffee}.\mathit{coin}.\mathit{coin}.\overline{\mathit{cup\_of\_coffee}}.\mathrm{CTM}$$

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

$$\begin{aligned}\mathrm{CS} \stackrel{\mathsf{def}}{=} & \quad \overline{\mathit{tea}}.\overline{\mathit{coin}}.\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS} \\ & + \quad \overline{\mathit{coffee}}.\overline{\mathit{coin}}.\overline{\mathit{coin}}.\mathit{cup\_of\_coffee}.\overline{\mathit{publish}}.\mathrm{CS}\end{aligned}$$

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

Example: Parallel composition

- On an average Tuesday morning the system

$$\mathrm{CTM} \mid \mathrm{CS}$$

is likely to behave as follows:

$$
\begin{array}{rl}
& (\mathit{tea}.\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM} + \mathit{coffee}.\mathit{coin}.\mathit{coin}.\overline{\mathit{cup\_of\_coffee}}.\mathrm{CTM}) \\
\mid & (\overline{\mathit{tea}}.\overline{\mathit{coin}}.\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS} + \overline{\mathit{coffee}}.\overline{\mathit{coin}}.\overline{\mathit{coin}}.\mathit{cup\_of\_coffee}.\overline{\mathit{publish}}.\mathrm{CS}) \\
\xrightarrow{\tau} & (\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM}) \mid (\overline{\mathit{coin}}.\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\tau} & (\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM}) \mid (\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\tau} & \mathrm{CTM} \mid (\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\overline{\mathit{teach}}} & \mathrm{CTM} \mid \mathrm{CS}
\end{array}
$$

- Note that the synchronisation of actions such as *tea* / $\overline{\mathit{tea}}$ 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 $\overline{a}$ from the interface of P
- Makes each name $a \in A$ and the corresponding co-name $\overline{a}$ inaccessible to the environment

Example: Restriction

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

$$\mathrm{CTM} \mid \mathrm{CS}$$

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

$$(\mathrm{CTM} \setminus \{\mathit{coffee}\}) \mid \mathrm{CS}$$

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

Summary: Syntax of CCS

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

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

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i \qquad Nil = 0 = \sum_{i \in \emptyset} P_i$$

CCS programs

- CCS program
  - A collection of defining equations of the form

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

  where $\mathrm{K} \in \mathcal{K}$ 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. $A \stackrel{\text{def}}{=} \overline{a}.A \mid A$
- 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

$a.b.\mathrm{A} + \mathrm{B}$ ✅

$(a.0 + \overline{a}.\mathrm{A}) \setminus \{a, b\}$ ✅

$(a.0 \,|\, \overline{a}.\mathrm{A}) \setminus \{a, \tau\}$ ❌

$\tau.\tau.\mathrm{B} + 0$ ✅

$(a.b.\mathrm{A} + \overline{a}.0) \,|\, \mathrm{B}$ ✅

$(a.b.\mathrm{A} + \overline{a}.0).\mathrm{B}$ ❌

Operational Semantics of CCS

Operational semantics

- Goal: Formalize the execution of a CCS process

Syntax		Semantics
CCS	→	LTS
(process term + equations)		(labelled transition systems)

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

$$\begin{tikzcd} & Q \arrow[dr, "\tau"] & \\ P \arrow[ur, "a"] \arrow[rr, "b"] & & R \end{tikzcd}$$

- 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: $\alpha.P \xrightarrow{\quad\alpha\quad} P$

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

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

- Recursion: $X \overset{\text{def}}{=} \cdots .\alpha.X$

  behavior:

$$
\begin{tikzcd}
X \arrow[r, bend left, "\cdots"] & \alpha.X \arrow[l, bend left, "\alpha"]
\end{tikzcd}
$$

Informal translation

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

behavior:

$$
\begin{tikzcd}
 & P \mid \beta.Q \arrow[dr, "\beta"] & \\
\alpha.P \mid \beta.Q \arrow[ur, "\alpha"] \arrow[dr, "\beta"] & & P \mid Q \\
 & \alpha.P \mid Q \arrow[ur, "\alpha"] &
\end{tikzcd}
$$

- 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 (sub-systems), P and Q, is an activity that is internal to $P \mid Q$
- Parallel composition: $\alpha.P \mid \beta.Q$
- Allows interaction if $\beta = \overline{\alpha}$

behavior:

$$\begin{tikzcd}
& P \mid \overline{a}.Q \arrow[dr, "\overline{a}"] & \\
a.P \mid \overline{a}.Q \arrow[ur, "a"] \arrow[rr, "\tau"] \arrow[dr, "\overline{a}"'] & & P \mid Q \\
& a.P \mid Q \arrow[ur, "a"'] &
\end{tikzcd}$$

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:

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

SOS rules for CCS

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

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

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

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

Example: Derivations

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

$$\text{COM1}\ \dfrac{\text{COM1}\ \dfrac{\text{CON}\ \dfrac{\text{ACT}\ \dfrac{}{a.A \xrightarrow{a} A}}{A \xrightarrow{a} A}\ A \stackrel{\text{def}}{=} a.A}{A \mid \overline{a}.0 \xrightarrow{a} A \mid \overline{a}.0}}{(A \mid \overline{a}.0) \mid b.0 \xrightarrow{a} (A \mid \overline{a}.0) \mid b.0}$$

Restriction and interaction

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

$$
\begin{tikzcd}
& a.0 \mid \overline{a}.0 \arrow[dl, "a"] \arrow[dr, "\overline{a}"] \arrow[dd, "\tau"] & \\
0 \mid \overline{a}.0 \arrow[dr, "\overline{a}"] & & a.0 \mid 0 \arrow[dl, "a"] \\
& 0 \mid 0 &
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
(a.0 \mid \overline{a}.0) \setminus \{a\} \arrow[d, "\tau"] \\
0 \mid 0
\end{tikzcd}
$$

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

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