Concurrency 3

CCS - Syntax and transitions, Equivalences

Catuscia Palamidessi
INRIA Futurs and LIX - Ecole Polytechnique

The other lecturers for this course:

Jean-Jacques Lévy (INRIA Rocquencourt)
James Leifer (INRIA Rocquencourt)
Eric Goubault (CEA)

http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2005/

Outline

Why a Calculus for Concurrency?

- The *Calculus for Communicating Systems* (CCS) was developed by R. Milner around the 80's.
- Other Process Calculi were proposed at about the same time: the *Theory of Communicating Sequential Processes* by T. Hoare and the *Algebra of Communicating Processes* by J. Bergstra and J.W. Klop.
- Researchers were looking for a calculus with few, orthogonal mechanisms, able to represent all the relevant concepts of concurrent computations. More complex mechanisms should be built by using the basic ones.
  - To help understanding / reasoning about / developing formal tools for concurrency.
  - To play a role, for concurrency, like that of the λ-calculus for sequential computation.

Inadequacy of standard models of computations

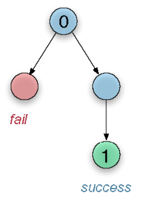
The λ calculus, the Turing machines, etc. are computationally complete, yet do not capture the features of concurrent computations like

- Interaction and communication
- Inadequacy of functional denotation
- Nondeterminism

Note: nondeterminism in concurrency is different from the nondeterminism used in Formal Languages, like for instance the Nondeterministic Turing Machines.

A few words about nondeterminism

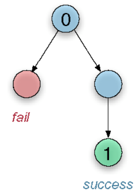
In standard computation theory, if we want to compute the partial function f s.t. $f(0) = 1$, a Turing Machine like this one is considered ok



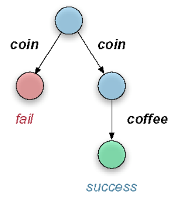
However, we would not be happy with a coffee machine that behaves in the same way

A few words about nondeterminism

In standard computation theory, if we want to compute the partial function f s.t. $f(0) = 1$, a Turing Machine like this one is considered ok



However, we would not be happy with a coffee machine that behaves in the same way



Nondeterminism in sequential models

- Convenient tool for solving certain problems in an easy way or for characterizing complexity classes (examples: search for a path in a graph, search for a proof etc.)
- Examples of nondeterministic formalisms:
  - The nondeterminismistic Turing machines
  - Logic languages like Prolog and λ Prolog
- The characteristics of nondeterminism in this setting:
  - It can be eliminated without loss of computational power by using backtracking.
  - Failures don't matter: all what we are interested on is the existence of succesful computations. A failure is reported only if all possible alternatives fail.

Nondeterminism in concurrent models

- Nondeterminism may arise because of interaction between processes.
- The characteristics of nondeterminism in this setting:
  - It cannot be avoided. At least, not without loosing essential parts of expressive power. All interesting models of concurrency cope with nondeterminism.
  - Failures do matter. Chosing the wrong branch might bring to an "undesirable situation". Backtracking is usually not applicable (or very costly), because the control is distributed: we should restart not one but several processes.
- Hence controlling nondeterminism is very important. In sequential programming is just a matter of efficiency, here is a matter of avoiding getting stuck in a wrong situation.

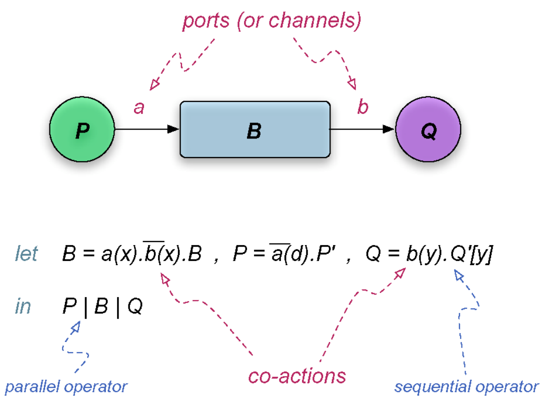
The basic kind of interaction (1/2)

- A calculus should contain only the primary constructs. For instance, the primary form of interaction. *But what is the primary form of interaction?*
- In general, concurrent languages can offer various kinds of communication. For instance:
  - Communications via shared memory.
  - Communication via channels.
  - Communication via broadcasting.
- and we could make even more distinctions
  - one-to-one / one-to-many
  - Ordered / unordered (i.e. queues / bags)
  - Bounded / unbounded.
- So what is the basic kind of communication?
- For CCS the answer was: *none of the above!*

The basic kind of interaction (2/2)

- In CCS, the fundamental model of interaction is *synchronous* and *symmetric*, i.e. the partners act at the same time performing complementary actions.
- This kind of interaction is called *handshaking*: the partners agree simoultaneously on performing the two (complementary) actions.
- In Java there is a separation between active objects (threads) and passive objects (resources). CCS avoids this separation: Every (non-elementary) entity is a process.
- For instance, consider two proceesses P and Q communicating via a buffer B. in CCS also B is a process and the communication is between P and B, and between Q and B.

Example: P and Q communicating via a buffer B



Syntax of CCS

- (channel, port) names: $a, b, c, \ldots$
- co-names: $\bar{a}, \bar{b}, \bar{c}, \ldots$ Note: $\bar{\bar{a}} = a$
- silent action: τ
- actions, prefixes: $\mu ::= a \mid \bar{a} \mid \tau$
- processes: $P, Q ::=$

$::=$	0	inaction
$\mid$	$\mu.P$	prefix
$\mid$	$P \mid Q$	parallel
$\mid$	$P + Q$	(external) choice
$\mid$	$(\nu a)P$	restriction
$\mid$	$\text{rec}_K P$	process P with definition $K = P$
$\mid$	K	(defined) process name

Labeled transition system

- The semantics of CCS is defined by in terms of a *labeled transition system*, which is a set of triples of the form

$$P \xrightarrow{\mu} Q$$

Meaning: *P evolves into Q by making the action* μ.

- The presence of the label μ allows us to keep track of the interaction capabilities with the environment.

Structural operational semantics

The transitions of CCS are defined by a set of inductive rules. The system is also called *structural semantics* because the evolution of a process is defined in terms of the evolution of its components.

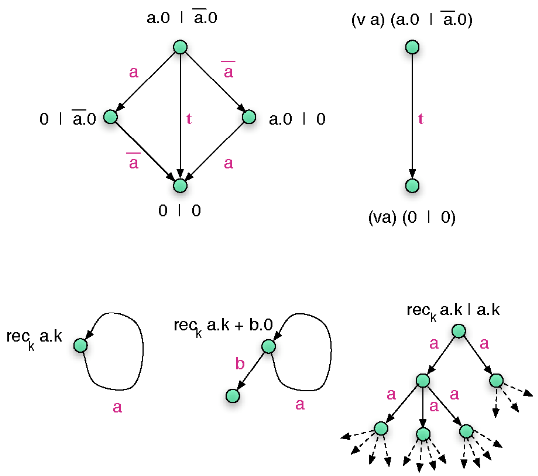
$$[\text{Act}]\ \frac{}{\mu.P \xrightarrow{\mu} P} \qquad [\text{Res}]\ \frac{P \xrightarrow{\mu} P' \quad \mu \neq a, \bar{a}}{(\nu a)P \xrightarrow{\mu} (\nu a)P'}$$

$$[\text{Sum1}]\ \frac{P \xrightarrow{\mu} P'}{P+Q \xrightarrow{\mu} P'} \qquad [\text{Sum2}]\ \frac{Q \xrightarrow{\mu} Q'}{P+Q \xrightarrow{\mu} Q'}$$

$$[\text{Par1}]\ \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \qquad [\text{Par2}]\ \frac{Q \xrightarrow{\mu} Q'}{P|Q \xrightarrow{\mu} P|Q'}$$

$$[\text{Com}]\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad [\text{Rec}]\ \frac{P[\text{rec}_K P/K] \xrightarrow{\mu} P'}{\text{rec}_K P \xrightarrow{\mu} P'}$$

Some examples



The restriction can be used to enforce synchronization

The parallel operator may cause infinitely many different states

The fragment of the calculus without parallel operator generates only finite automata / regular trees

Motivation

- It is important to define formally when two system can be considered equivalent
- There may be various "interesting" notion of equivalence, it depends on what we want (which observables we want to preserve)
- A good notion of equivalence should be a congruence, so to allow modular verification

Examples: possible definitions of a coffee machine

- $\mathrm{rec}_K\, coin.(coffee.\overline{ccup}.K + tea.\overline{tcup}.K)$
- $coin.\mathrm{rec}_K(coffee.\overline{ccup}.coin.K + tea.\overline{tcup}.coin.K)$
- $\mathrm{rec}_K(coin.coffee.\overline{ccup}.K + coin.tea.\overline{tcup}.K)$
- Question: which of these machines can we safely consider equivalent?
- Note that these machines have all the same traces.

Exercises

- Define in CCS a semaphore with initial value n
- Show that the trace equivalence is not a congruence in CCS. By traces here we mean the complete (finite or infinite) traces of all possible runs.