Lecture 1: Welcome/Introduction/Overview

Concurrency Theory

Summer 2024

Dr. Stephan Mennicke

April 8th, 2024

TU Dresden, Knowledge-Based Systems Group

Welcome to *Concurrency Theory* in Summer 2024

Welcome to *Concurrency Theory* in Summer 2024.

Organizational Issues

Room, Time, URL

1. **Sessions:**
   Tuesdays DS2 (9:20–10:50) in APB/E005 ⇝ planned as *lectures*
   Wednesdays DS3 (11:10–12:40) in APB/E005 ⇝ planned as *exercises*
2. **No sessions:**
   - May 1 (labor day) *Wednesday*
   - May 21 & 22 (Pentecost week)
   - June 5 (dies academicus) *Wednesday*
3. **Website:**
   https://iccl.inf.tu-dresden.de/web/Concurrency_Theory_(SS2024)
   (slides, literature, etc.)
4. **Matrix (Chat):**
   https://matrix.to/#/#concur:tu-dresden.de
5. **Examination:**
   - *oral examination* (20-25 minutes)
   - registration depending on your study/exam regulations

Introduction

What is Concurrency Theory?

Concurrency Theory = study of the *semantics* of *concurrent* languages or systems.

⇝ several activities (the *processes*) may run concurrently

Central Questions:

1. What is a *process*, mathematically?
2. What does it mean for two processes to be *equal*?
    - seek notions of equality that are effective
    - equality must be justifiable, according to the notion of *process*

From Functions to Processes

First, **no** concurrency = sequential (programming) languages

Main Tool (since the 1970s):

denotational semantics = programs are functions

- concepts are clear for functional languages (e.g., the λ-calculus)
- also applicable to imperative languages

  How? $\rightsquigarrow$ next lectures

Example 1: Consider the following two program fragments:

`X := 2` and `X := 1; X := X+1`

They yield the same *function* f. Informally, if f is applied to store $s : \mathrm{Var} \to \mathbb{N}$, then store $s' : \mathrm{Var} \to \mathbb{N}$ is produced such that $s'(X) = 2$ and $s'(X) = s(Y)$ for all variables $Y \neq X$.

Consequence: Programs are *equal* if they have the same denotation.

Now with Concurrency

Examples are troublesome in languages with concurrency features:
$P \parallel Q$ means program P runs concurrently with program Q.

$\rightsquigarrow$ intuitively, $P \parallel Q$ is the *parallel composition* of P and Q

Example 2: Now consider the programs in the *parallel context* $[\cdot] \parallel$ `X := 2`. Then

$$\texttt{X := 2} \parallel \texttt{X := 2}$$

always terminates with `X` set to 2 while

$$\texttt{(X := 1; X := X+1)} \parallel \texttt{X := 2}$$

may terminate with values different from 2.

Consequence 1: Function equality is not preserved by *parallel composition*.

Consequence 2: Parallel programs are not functions, they are *processes*.

The Goal is Compositionality

Why Compositionality?

- allows to exploit the structure of the language for reasoning
- inference of properties from components to larger systems
- optimization of program components

We aim for a *compositional semantics*.

On the level of equality (i.e., equivalences) we are looking for *congruences*.

More Issues with Programs as Functions

Nontermination as a Feature

- concurrent programs may not terminate and yet produce meaningful computations
  - operating systems
  - controllers of railway stations
- in sequential languages, programs that do not terminate have *mathematically* undefined (i.e., undesirable or wrong) behavior

Nondeterminism

- nondeterministic behavior is everywhere in concurrent systems
- sequential languages use powerset or powerdomain constructions

  $\rightsquigarrow$ quickly becomes cumbersome
- if nondeterminism is a language feature, then we cannot distinguish it from concurrency features

Interaction is Key

The example programs

`X := 2` and `X := 1; X := X+1`

should be distinguished because of their *interaction* with the memory. The difference between them is harmless in sequential languages: Why?

$\rightsquigarrow$ only initial and final state are visible to the outside world (i.e., the *environment*).

New keyword: *interaction*

- computation = interaction (in concurrency)
- examples:
  - access to memory cells
  - queries to databases
  - selection of a beverage at a vending machine

Processes, Behavior, and Interaction

- participants in interactions are called *processes*
- the *behavior* of a process should tell
  - the *When?*
  - and *How?* of interaction with the environment
- need a mathematically precise model of behavior
- interaction is kept simple: handshake synchronization

Another Example: A Vending Machine

The *vending machine* is capable of dispensing coffee or tea for 1€.

- it has a slot for the coin (1€);
- it has a button for picking coffee (c);
- and another button for requesting tea (t)

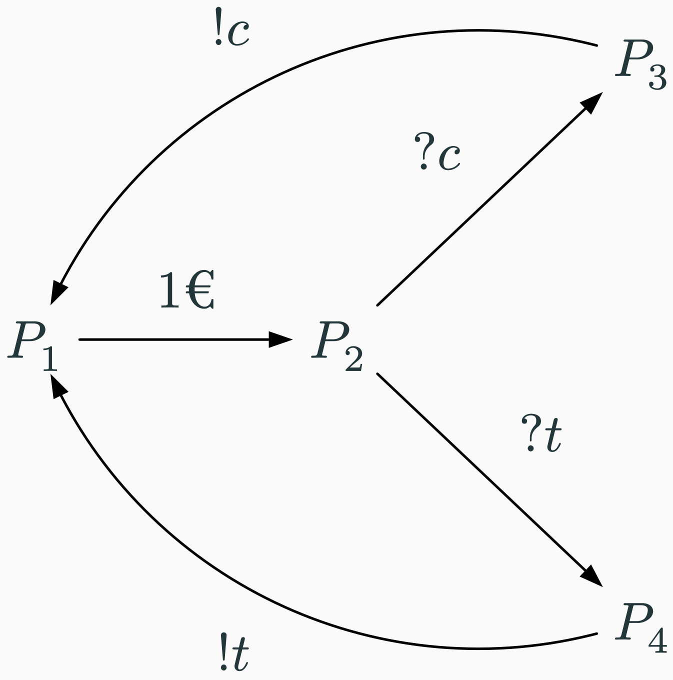
The behavior of the vending machine is described by what we can observe by interaction

⇝ experiments

The **main model** we use throughout the lecture is that of *labeled transition systems* (LTS):

- state information
- for each state, what interactions are possible

The Vending Machine as an LTS



- states are P_1, P_2, P_3, and P_4
- labeled edges (i.e., *transitions*) tell us about the possible interactions

Labeled Transition Systems

Definition 3 (Labeled Transition System): A *labeled transition system* (LTS) is a triple $(\mathrm{Pr}, \mathrm{Act}, \rightarrow)$ where Pr is a non-empty set, the *domain* of the LTS; Act is the set of *actions*; and $\rightarrow \subseteq \mathrm{Pr} \times \mathrm{Act} \times \mathrm{Pr}$ is the *transition relation*.

The elements of Pr are sometimes called *states*, more often *processes*.

Processes range over by $P, P_1, P_2, \ldots$ and Q or R, actions usually by $\mu, \mu_1, \mu_2, \ldots$.

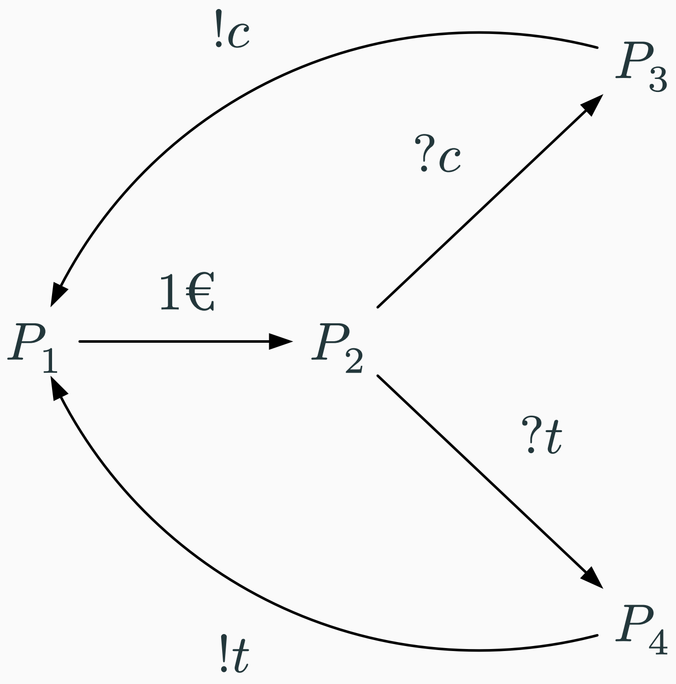
We write $P \xrightarrow{\mu} Q$ for $(P, \mu, Q) \in \rightarrow$.

For every action $\mu \in \mathrm{Act}$, $\xrightarrow{\mu}$ is a binary relation over Pr.

If $s = \mu_1 \mu_2 \ldots \mu_k$, then $P \xrightarrow{s} P'$ if there are $P_1, P_2, \ldots, P_{k-1} \in \mathrm{Pr}$ such that $P \xrightarrow{\mu_1} P_1$, $P_1 \xrightarrow{\mu_2} P_2$, …, $P_{k-1} \xrightarrow{\mu_k} P'$.

Write $P \xrightarrow{\mu}$ if there is a P' such that $P \xrightarrow{\mu} P'$ and $P \overset{\mu}{\nrightarrow}$ if there is none.

The Vending Machine as an LTS



This is the LTS $V = (\mathrm{Pr}, \mathrm{Act}, \rightarrow)$ where $\mathrm{Pr} = \{P_1, P_2, P_3, P_4\}$, $\mathrm{Act} = \{1€, ?c, ?t, !c, !t\}$, and $\rightarrow = \{(P_1, 1€, P_2), (P_2, ?c, P_3), (P_2, ?t, P_4), (P_3, !c, P_1), (P_4, !t, P_1)\}$

Classes of LTSs and Processes

Definition 4 (Induced LTS): Let L be an LTS and P a process of L. The *induced LTS* by P is the smallest LTS L' (with domain Pr) such that $P \in \mathrm{Pr}$, Act is the same action set as for L, and if $Q \in \mathrm{Pr}$ and there is a transition $Q \xrightarrow{\mu} Q'$ in L, then $Q' \in \mathrm{Pr}$ and $Q \xrightarrow{\mu} Q'$ is a transition of L'.

Definition 5 (LTS Classes): An LTS is

- *image-finite* if for each μ, relation $\xrightarrow{\mu}$ is image-finite (i.e., for all P, the set $\left\{ P' \,\middle|\, P \xrightarrow{\mu} P' \right\}$ is finite);
- *finitely branching* if it is image-finite and, for each P, the set $\left\{ \mu \,\middle|\, P \xrightarrow{\mu} \right\}$ is finite;
- *finite-state* if it has a finite number of states;
- *finite* if it is finite-state and acyclic;
- *deterministic* if all processes are deterministic (i.e., for P and μ, $P \xrightarrow{\mu} P_1$ and $P \xrightarrow{\mu} P_2$ implies $P_1 = P_2$)

Summary and Overview

Summary

- denotations as sound basis for *sequential* programming language semantics
- denotations insufficient when concurrency is involved
  - computation is interaction
  - interaction between processes
- labeled transition systems (Definition 3) as **the** model for behavior
  - basic notions and notations
  - classes of LTSs and processes (Definition 5)

Overview

Part 0: Completing the Introduction (next)

- learning about *bisimilarity* and *bisimulations*

Part 1: Semantics of (Sequential) Programming Languages

- WHILE – an old friend
- denotational semantics (a baseline and an exercise of the inductive method)
- natural semantics and (structural) operational semantics

Part 2: Towards Parallel Programming Languages

- bisimilarity and its success story
- deep-dive into induction and coinduction
- algebraic properties of bisimilarity

Part 3: Expressive Power

- Calculus of Communicating Systems (CCS)
- Petri nets

Literature

- Sangiorgi, D. (2012). **Introduction to bisimulation and coinduction**. Cambridge University Press.
- Esparza, J. **Petri Nets Lecture Notes** from a course given at TU Munich https://archive.model.in.tum.de/um/courses/petri/SS2019/PNSkript.pdf
- Reisig, W. (2013). **Understanding Petri Nets**. Springer Berlin Heidelberg.
- Sangiorgi, D., & Walker, D. (2003). **The pi-calculus: a theory of mobile processes**. Cambridge University Press.
- Milner, R. (1980). **A calculus of communicating systems**. Springer Berlin Heidelberg.
- Milner, R. (1999). **Communicating and mobile systems**. Cambridge University Press.
- Davide Sangiorgi (2012). **Advanced topics in bisimulation and coinduction**. Cambridge University Press.