Programming Language Theory

Concurrency Calculi

Ralf Lämmel

This lecture is based on a number of different resources as indicated per slide.
Concurrency

What is concurrency?

What makes concurrent programming different from sequential programming?

What are the core components of a concurrent language?
Concurrency

• Possible inter-thread communication mechanisms:
  • Read/write to shared memory.
  • Locks.
  • Monitors (a.k.a. wait/notify).
  • Buffered streams.
  • Unbuffered streams.
  • ...

• Which of these does Java support?
• Which should we include in a foundational calculus?
History

- Models of concurrency (late 1970s-80s): Communicating Sequential Processes (Hoare), Petri Nets (Petri), Calculus of Communicating Systems (Milner), ...

- Additional features to model dynamic network topologies (late 1980s-90s): Pi-calculus (Milner), Higher order pi-calculus (Sangiorgi), Ambients (Cardelli and Gordon), ...
In need of designated calculi
Program meanings

Program Meanings =
Memories \rightarrow \text{Memories}.

Program Meanings =
Memories \rightarrow P(\text{Memories})

Ok for sequential programs

Needed for non-deterministic programs
Parallelism and shared memory

Program \( P_1 : x := 1 \); \( x := x + 1 \)
Program \( P_2 : x := 2 \)

Semantics\( (P_1) = \text{Semantics}(P_2) \)
Parallelism and shared memory

Program $P_1 : x := 1 ; x := x + 1$
Program $P_2 : x := 2$
Program $Q : x := 3$

Program $R_1 : P_1 \text{ par } Q$
Program $R_2 : P_2 \text{ par } Q$

Semantics($R_1$) \neq \text{Semantics}(R_2)$
“Once the memory is no longer at the behest of a single master, then the master-to-slave (or: function-to-value) view of the program-to-memory relationship becomes a bit of a fiction. An old proverb states: He who serves two masters serves none. It is better to develop a general model of interactive systems in which the program-to-memory interaction is just a special case of interaction among peers.”
The shared memory model
Memory as an interactive process

Program variables as channels

Process

Process
Memory as a distributed process

Memory cells are processes.

Memories are no longer monolithic.
The Calculus of Communicating Systems
Agents and ports

- **Agent** \( C \)
  - Dynamic system is network of *agents*.
  - Each agent has own identity persisting over time.
  - Agent performs *actions* (external communications or internal actions).
  - *Behavior* of a system is its (observable) capability of communication.

- Agent has labeled *ports*.
  - Input port \( \text{in} \).
  - Output port \( \text{out} \).
A simple example

Behavior of $C$:

- $C := \text{in}(x).C''(x)$
- $C''(x) := \text{out}(x).C$

Process behaviors are described as (mutually recursive) equations.
Example: bounded buffers

Bounded buffer $Buff_n(s)$

- $Buff_n(\langle \rangle) := in(x).Buff_n(\langle x \rangle)$
- $Buff_n(\langle v_1, \ldots, v_n \rangle) :=$
  $\overline{out}(v_n).Buff_n(\langle v_1, \ldots, v_{n-1} \rangle)$
- $Buff_n(\langle v_1, \ldots, v_k \rangle) :=$
  $\overline{in}(x).Buff_n(\langle x, v_1, \ldots, v_k \rangle)$
  $+ \overline{out}(v_k).Buff_n(\langle v_1, \ldots, v_{k-1} \rangle)(0 < k < n)$
Used language elements

• Basic combinator ’+’
  - \( P + Q \) behaves like \( P \) or like \( Q \).
  - When one performs its first action, other is discarded.
  - If both alternatives are allowed, selection is non-deterministic.

• Combining forms
  - Summation \( P + Q \) of two agents.
  - Sequencing \( \alpha.P \) of action \( \alpha \) and agent \( P \).

Process definitions may be parameterized.
Example: a vending machine

- Big chocolade costs 2p, small one costs 1p.
- $V := 2p.\text{big}.\text{collect}.V$
  + $1p.\text{little}.\text{collect}.V$

Exercises:
Identify input vs. output.
What behaviors make sense for users?
Example: a multiplier

- Twice := in(x).out(2 * x).Twice.
- Output actions may take expressions.
Example: The JobShop

A simple production line:

- Two people (the jobbers).
- Two tools (hammer and mallet).
- Jobs arrive sequentially on a belt to be processed.

Ports may be linked to multiple ports:

- Jobbers compete for use of hammer.
- Jobbers compete for use of job.
- Source of non-determinism.

Ports of belt are omitted from system:

- in and out are external.
- Internal ports are not labelled:
  - Ports by which jobbers acquire and release tools.

These slides were obtained by copy&paste&edit from W. Schreiner’s concurrency lectures (Kepler University, Linz).
Example: The JobShop

- A simple production line:
  - Two people (the jobbers).
  - Two tools (hammer and mallet).
  - Jobs arrive sequentially on a belt to be processed.

- Ports may be linked to multiple ports.
  - Jobbers compete for use of hammer.
  - Jobbers compete for use of job.
  - Source of non-determinism.

- Ports of belt are omitted from system.
  - in and out are external.

- Internal ports are not labelled:
  - Ports by which jobbers acquire and release tools.
The tools of the JobShop

• Behaviors:
  - Hammer := geth.Busyhammer
    Busyhammer := puth.Hammer
  - Mallet := getm.Busymallet
    Busymallet := putm.Mallet

• Sort = set of labels
  - $P : L$ ... agent $P$ has sort $L$
  - Hammer: \{geth, puth\}
    Mallet: \{getm, putm\}
    Jobshop: \{in, out\}
The jobbers of the JobShop

• Different kinds of jobs:
  – Easy jobs done with hands.
  – Hard jobs done with hammer.
  – Other jobs done with hammer or mallet.

• Behavior:
  – Jobber := in(job).Start(job)
  – Start(job) := if easy(job) then Finish(job)
    else if hard(job) then Uhammer(job)
    else Usetool(job)
  – Usetool(job) := Uhammer(job)+Umallet(job)
  – Uhammer(job) := geth.puth.Finish(job)
  – Umallet(job) := getm.putm.Finish(job)
  – Finish(job) := out(done(job)).Jobber
Composition of the agents

- **Jobber-Hammer subsystem**
  - Jobber | Hammer
  - Composition operator |
  - Agents may proceed independently or interact through complementary ports.
  - Join complementary ports.

- **Two jobbers sharing hammer:**
  - Jobber | Hammer | Jobber
  - Composition is commutative and associative.
Further composition

• **Internalisation** of ports:
  – No further agents may be connected to ports:
  – *Restriction* operator \( L \)
  – \( L \) internalizes all ports \( L \).
  – \((\textit{Jobber} \mid \textit{Jobber} \mid \textit{Hammer}) \setminus \{\text{geth}, \text{puth}\}\)

• **Complete system:**
  – \textit{Jobshop} := \((\textit{Jobber} \mid \textit{Jobber} \mid \textit{Hammer} \mid \textit{Mallet}) \setminus L\)
  – \( L := \{\text{geth}, \text{puth}, \text{getm}, \text{putm}\}\)
“... sequential composition is indeed a special case of parallel composition ... in which the only interaction between occurs when $P$ finishes and $Q$ begins ...”

\[ P; Q \quad \text{not part of CCS} \]

\[ P|Q \quad \text{part of CCS} \]
Reformulations

- **Relabelling Operator**
  - $P[l'_1/l_1, \ldots, l'_n/l_n]$
  - $f(l) = f(l')$

- **Semaphore agent**
  - $Sem := get.put.Sem$

- **Reformulation of tools**
  - $Hammer := Sem[geth/get, puth/put]$
  - $Mallet := Sem[getm/get, putm/put]$

These slides were obtained by copy&paste&edit from W. Schreiner’s concurrency lectures (Kepler University, Linz).
In need of equality of agents

- **Strongjobber** only needs hands:
  - $\text{Strongjobber} \equiv \text{in}(\text{job}).\overline{\text{out}}(\text{done}(\text{job})).\text{Strongjobber}$

- Claim:
  - $\text{Jobshop} = \text{Strongjobber} \mid \text{Strongjobber}$
  - Specification of system Jobshop
  - Proof of equality required.

In which sense are the processes equal?
Formalization of CCS
The core calculus

No value transmission: just synchronization

• Names and co-names
  – Set $A$ of *names* (geth, ackin, ...)
  – Set $\overline{A}$ of *co-names* (\overline{geth}, \overline{ackin}, ...)
  – Set of *labels* $L = A \cup \overline{A}$

• Actions
  – *Completed (perfect)* action $\tau$.
  – $Act = L \cup \{\tau\}$

• Transition $P \xrightarrow{l} Q$ with action $l$
  – *Hammer* geth $\rightarrow$ *Busyhammer*
Transition rules of the core calculus

- Act \[ \alpha.E \xrightarrow{\alpha} E \]
- Sum \[ j \] \[ E_j \xrightarrow{\alpha} E_j' \] \[ \sum E_i \xrightarrow{\alpha} E_i' \]
- Com\(_1\) \[ E \xrightarrow{\alpha} E' \] \[ E|F \xrightarrow{\alpha} E'|F \]
- Com\(_2\) \[ F \xrightarrow{\alpha} F' \] \[ E|F \xrightarrow{\alpha} E'|F' \]
- Com\(_3\) \[
\begin{align*}
\text{Rule 3:} & \quad E \xrightarrow{l} E' \quad F \xrightarrow{\tau} F' \\
\text{Result:} & \quad E|F \xrightarrow{\tau} E'|F'
\end{align*}
\]
- Res \[ E \xrightarrow{\alpha} E' \] \[ E \setminus L \xrightarrow{\alpha} E' \setminus L \] \[ (\alpha, \overline{\alpha} \text{ not in } L) \]
- Rel \[ E \xrightarrow{\alpha} E' \] \[ E[f] \xrightarrow{f(\alpha)} E'[f] \]
- Con \[ P \xrightarrow{\alpha} P' \] \[ A \xrightarrow{\alpha} P' \] \[ (A := P) \]

This rule makes clear that no more than two agents participate in communication.

This rule rules out transitions with hidden names.

This is about the application of definitions for agents.
The value-passing calculus

• Values passed between agents
  – Can be reduced to basic calculus.
  – \( C' := \text{in}(x).C''(x) \)
  \( C''(x) := \text{out}(x).C \)
  – \( C := \sum_v \text{in}_v.C'_v \)
  \( C'_v := \text{out}_v.C \ (v \in V) \)
  – Families of ports and agents.

• The full language
  – Prefixes \( a(x).E, \bar{a}(e).E, \tau.E \)
  – Conditional if \( b \) then \( E \)

• Translation
  – \( a(x).E \Rightarrow \sum_v.E\{v/x\} \)
  – \( \bar{a}(e).E \Rightarrow \bar{a}_e.E \)
  – \( \tau.E \Rightarrow \tau.E \)
  – if \( b \) then \( E \Rightarrow (E, \text{if } b \text{ and } 0, \text{otherwise}) \)
Bisimulation
(very informally)

- Two agent expressions $P$, $Q$ are bisimilar:
  - If $P$ can do an $\alpha$ action towards $P'$,
  - then $Q$ can do an $\alpha$ action towards $Q'$,
  - such that $P'$ and $Q'$ are again bisimilar,
  - and v.v.

Intuitively two systems are bisimilar if they match each other's moves. In this sense, each of the systems cannot be distinguished from the other by an observer. [Wikipedia]
Laws
Summation laws

\[
\begin{align*}
- P + Q &= Q + P \\
- P + (Q + R) &= (P + Q) + R \\
- P + P &= P \\
- P + 0 &= P
\end{align*}
\]
• **Composition laws**
  
  - $P|Q = Q|P$
  - $P|(Q|R) = (P|Q)|R$
  - $P|0 = P$

• **Restriction laws**
  
  - $P\setminus L = P$, if $L(P) \cap (L \cup \overline{L}) = \emptyset$.
  - $P\setminus K\setminus L = P\setminus (K \cup L)$
  - \ldots

• **Relabelling laws**
  
  - $P[\text{id}] = P$
  - $P[f][f'] = P[f' \circ f]$
  - \ldots
Non-laws

• $\tau.P = P$
  
  - $A = a.A + \tau.b.A$
  
  - $A' = a.A' + b.A'$
  
  - $A$ may switch to state in which only $b$ is possible.
  
  - $A'$ always allows $a$ or $b$.

• $\alpha.(P + Q) = \alpha.P + \alpha.Q$
  
  
  - $b.P$ is $a$-derivative of right side, not capable of $c$ action.
  
  - $a$-derivative of left side is capable of $c$ action!
  
  - Action sequence $a, c$ may yield deadlock for right side.
Pi-calculus

A minimal model
with ‘enough stuff’ to perform interesting computation.
(Computations more powerful than the lambda-calculus).
Pi calculus

http://en.wikipedia.org/wiki/%CE%A0-calculus

First shot:

\[ P, Q, R ::= \]

- \( 0 \)
- \( \text{out } x \ y; \ P \)
- \( \text{in } x \ (y); \ P \)
- \( P \parallel Q \)

- Completed process
- Output prefixing: emit \text{name } y on channel \ x
- Input prefixing: wait for a name on channel \ x to be bound to \ y
- Concurrency

Ralf Lämmel: Programming Language Theory Lecture, 2011, University of Koblenz-Landau
Example programs

1. `out stdout hello; out stdout world; 0`
2. `in stdin (name); out stdout hello; out stdout name; 0`
3. `(out c fred; 0) | (in c (name); out d name; 0)`
4. `(out c fred; out c wilma; 0) | (in c (x); out d x; 0) | (in c (y); out e y; 0)`
5. `(out c fred; in d x; 0) | (in c (y); out d wilma; 0)`
6. `(in d x; out c fred; 0) | (in c (y); out d wilma; 0)`
7. `(out c fred; in d (x); 0) | (out d wilma; in c (y); 0)`

What do these programs do?
Dynamic semantics

Structural congruence $P \equiv Q$ is generated by:

1. If $P =_{\alpha} Q$ then $P \equiv Q$.
2. $P \mid Q \equiv Q \mid P$.
3. $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$.

Dynamic semantics $P \rightarrow Q$ is generated by:

1. $(\text{out } x \ y; \ P) \mid (\text{in } x \ (z); \ Q) \rightarrow P \mid Q[ y / z ]$
2. If $P \rightarrow Q$ then $P \mid R \rightarrow Q \mid R$.
3. If $P \equiv \rightarrow \equiv Q$ then $P \rightarrow Q$. 
Recursion? Looping? Infinite Behavior?

Minimal solution replication: !P ‘acts like’ P | P | P | ...
Creation of new channels

Minimal solution *channel generation:* \( \text{new } (x); P \) generates a fresh channel for use in \( P \).

Example:

1. \( \text{new } (c); \text{out } x \ c; \text{in } c \ (y_1); \ldots \text{in } c \ (y_n); P \)
2. \( \text{in } x \ (c); \text{out } c \ z_1; \ldots \text{out } c \ z_n; Q \)

Put these in parallel, and what happens?

New channel generation acts a lot like new object generation / new key generation / new nonce generation / ...

Dynamic semantics just given by:

\[
\text{(new } (x); P) \parallel Q \equiv \text{new } (x); (P \parallel Q) \quad \text{(as long as } x \not\in Q)\]

If \( P \rightarrow Q \) then \( \text{new } (x); P \rightarrow \text{new } (x); Q. \)
Derived forms

Multiple messages:

\[
\text{in } x \ (y_1,\ldots,y_n); \ P \\
= \text{new } (c); \ \text{out } x \ c; \ \text{in } c \ (y_1); \ .. \ \text{in } c \ (y_n); \ P
\]

\[
\text{out } x \ (z_1,\ldots,z_n); \ Q \\
= \text{in } x \ (c); \ \text{out } c \ z_1; \ .. \ \text{out } c \ z_n; \ Q
\]

Let’s double check:

\[
( \text{in } x \ (y_1,\ldots,y_n); \ P \mid \text{out } x \ (z_1,\ldots,z_n); \ Q ) \rightarrow^* \\
P[z_1/y_1,\ldots,z_n/y_n] \mid Q
\]
In need of garbage collection

new (c); $P =_{gc} P$ (when $c \notin P$)

new (c); in $c$ (x); $P =_{gc} 0$

new (c); !in $c$ (x); $P =_{gc} 0$

new (c); out $c$ x; $P =_{gc} 0$

new (c); !out $c$ x; $P =_{gc} 0$

$P | 0 =_{gc} P$

Let’s double check:

( in $x$ ($y_1$, ..., $y_n$); $P | out x$ ($z_1$, ..., $z_n$); $Q$ )

$\rightarrow^* =_{gc} P[z_1/y_1, ..., z_n/y_n] | Q$
Correctness of garbage collection:

If $P =_{gc} Q$ and $P \rightarrow P'$
then $P' =_{gc} Q'$ and $Q \rightarrow Q'$
More derived forms

Booleans:

True(b)
   = !in b (x, y); out x (); 0

False(b)
   = !in b (x, y); out y (); 0

if (b) { P } else { Q }
   = new (t); new (f); ( out b (t, f); 0 | in t (); P | in f (); Q )

Sanity check:

True(b) | if (b) { P } else { Q }
   →* =gc True(b) | P
Many derived forms

Can also code integers, linked lists, ...

and the lambda-calculus...

and concurrency controls like mutexes, mvars, ivars, buffers, etc.
• **Summary**: CCS and Pi-calculus
  - Modeling systems of interacting processes using channels.
  - Approach amenable to formal analysis.
  - Equivalence is based on communication behavior.

• **Recommended reading**:
  - Milner’s “Elements of Interaction”
  - CCS tutorial [AcetoLI05]

• **Outlook**:
  - End Prolog-driven section of this course
  - Begin Haskell-driven section
  - (Preparation of) Midterm