Concurrency:
Languages, Programming and Theory

– Pi Calculus –

Session 6(&7) – November 26, 2003

Uwe Nestmann

EPFL-LAMP

Redexes

There are (at least) two reasons for studying LTSs
(as opposed to mere reductions as in the λ-calculus):

- the emphasis in on *interaction* with other programs
- *redexes*$^{(*)}$ in a concurrent program are usually distributed over terms, not juxtaposed as in λ-calculus.

$$(\lambda x.E)F \qquad\qquad \overline{a}\langle v\rangle.P_1 \mid (\ P_2 \mid a(x).P_3\ )$$

$(*)$ ***redexes*** are the “source” of reductions or internal transitions, visible as the pattern in the conclusion of either the β-rule (in λ) or the COMM-rule (in [VP]CCS).

Unbounded Structures: Stacks (I)

- “specification”:
  stored values are encoded in the index of the identifier

$$\frac{\begin{array}{lcl} \vec{n} & := & \{\ \text{empty}, \text{push}, \text{pop}\ \} \\ \vec{v} & \in & \mathcal{V}^* \end{array}}{\begin{array}{lcl} \multicolumn{3}{l}{\text{Stack}_{\vec{w}}(\ \vec{n}\ )} \\ \text{Stack} & \stackrel{\text{def}}{=} & \text{push}(x).\text{Stack}_x + \overline{\text{empty}}\langle\rangle.\text{Stack} \\ \text{Stack}_{v,\vec{w}} & \stackrel{\text{def}}{=} & \text{push}(x).\text{Stack}_{x,v,\vec{w}} + \overline{\text{pop}}\langle v\rangle.\text{Stack}_{\vec{w}} \end{array}}$$

- needs an unbounded number of process identifiers . . .
- does not exploit “concurrency inside”

Unbounded Structures: Stacks (II)

- "implementation":
  using a **chain** of individual cells for the stored values
- cells can have one of the following states:
  - E: nothing is left in the stack accessible through this cell
  - C_v: a cell containing value v
  - D: nothing left in this particular cell,
    but maybe "beyond" D (i.e., on "the right of it").

Unbounded Structures: Stacks (III)

$$\vec{n} \quad := \quad \text{push, empty, pop, not, drop, pull}$$

$$X\langle\,\vec{n}\,\rangle \frown Y\langle\,\vec{n}\,\rangle \quad := \quad (\boldsymbol{\nu} a, b, c)$$

$$(\,X\langle\,\vec{n}\,\rangle[{}^{a,b,c}/_{\text{not,drop,pull}}] \mid Y\langle\,\vec{n}\,\rangle[{}^{a,b,c}/_{\text{push,empty,pop}}]\,)$$

$$E \quad := \quad \text{push}(x).(C_x \frown E) + \overline{\text{empty}}\langle\rangle.E$$

$$C_v \quad := \quad \text{push}(x).(C_x \frown C_v) + \overline{\text{pop}}\langle v\rangle.D$$

$$D \quad := \quad \text{pull}(x).C_x + \text{drop}().E$$

$$S_{\vec{v}} \quad := \quad C_{v_1} \frown \cdots \frown C_{v_n} \frown E$$

- Calculate the states for the transition sequence

  $$\xrightarrow{\text{push}1} \xrightarrow{\text{push}2} \xrightarrow{\text{pop}2}$$

  and “stabilize” the result (by running possible τ-transitions).
- Compare $\text{Stack}_{\vec{v}}$ and $S_{\vec{v}}$ . . .

Turing Power

A **Turing-machine** consists of:

- a finite alphabet of symbols
- an infinite tape
- a finite control mechanism
- movement or r/w-head to left or right

A Turing-machine can be nicely simulated with concurrent processes by two stacks (the tape). Neither an infinite alphabet nor infinite summation is necessary for this. [Milner 89]

1. CCS is *Turing-powerful*.
2. The halting problem for some “Turing machine” TM can be related to the *existence* of an infinite sequence TM $\longrightarrow^{\omega}$.

Unbounded Structures: Stacks (IV)

Some criticism:

- D's cannot be reused for storing *new* values (neither inner nor outer D's!).
- E's are never "used", pile up and stay around. (Note that, although $E \frown E$ "=" E, explicit garbage collection would be required.)

Unbounded Structures: Stacks (V)

$$
\begin{array}{rcl}
E & := & \mathsf{push}(x).C_x + \overline{\mathsf{empty}}\langle\rangle.E \\
C_v & := & \mathsf{push}(x).(C_x \frown C_v) + \overline{\mathsf{pop}}\langle v\rangle.D + \overline{\mathsf{not}}\langle v\rangle.D \\
D & := & \mathsf{pull}(x).C_x + \mathsf{drop}().E + \mathsf{push}(x).C_x \\
\hline
S_{\vec{v}} & := & C_{v_1} \frown \cdots \frown C_{v_n} \frown E
\end{array}
$$

- ☐ What are the problems of this "implementation"?
    -
    -
    -

Expressiveness

Although Turing-powerful,
CCS is—in some particular sense—not expressive enough:

---

it is not possible to *cut out unusable (=dead) cells* E.

If we had the possibility to *dynamically change the interconnection structure* among process components, then cells could *drop out* by connecting their left and right neighbors together.

One way to do this is the transmission of "*channels over channels*".

Name-Passing Syntax

negative actions $\overline{a}\langle v\rangle$: *<u>send name</u>* v *over name* a*.*

positive actions $a(x)$: *<u>receive any name</u>, say* v*, over name* a
and "bind the result" to name x*.*

Binding results in *substitution*
of the formal parameter x by the actual parameter v.

polyadic communication $\overline{a}\langle\vec{v}\rangle$ and $a(\vec{x})$ ($\vec{x}$ pairwise different)
transmit many values at a time.

Syntax Conventions

$\mathcal{N}$	names	$a, b, c \ldots, x, y, z$
$\mathcal{A}$	actions	$\pi \ ::= \ x(y) \ \mid \ \overline{x}\langle y \rangle \ \mid \ \tau$

- finite sequences $\vec{a} \ldots$
- All values/variables/channels are just names.
  Parentheses usually indicate bindings.
  Angled brackets are often omitted.
- construct for the ***replication*** **!** P of processes P
- ***parametric processes*** with defining equations
  are modeled via the more primitive notion
  of replication and name-passing

Pi Calculus

Definition: The set $\mathcal{P}$ of π-calculus proc. exp. is defined (precisely) by the following syntax:

$$
\begin{array}{rcl}
P & ::= & M \quad | \quad P|P \quad | \quad (\boldsymbol{\nu} a)\, P \quad | \quad \boxed{\mathbf{!}\, P} \\
 & | & A\langle\, \vec{a}\,\rangle
\end{array}
$$

$$
M \;::=\; \mathbf{0} \quad | \quad \boxed{\pi.P} \quad | \quad M + M
$$

We use P, Q, P_i to stand for process expressions.

- $(\boldsymbol{\nu} ab)\, P$ abbreviates $(\boldsymbol{\nu} a)\,(\boldsymbol{\nu} b)\, P$
- $\sum_{i\in\{1..n\}} \pi_i.P_i$ abbreviates $\pi_1.P_1 + \ldots + \pi_n.P_n$

Mobility ? “Flowgraphs” !

$P {=} \overline{x}\langle z \rangle.P'$

$Q {=} x(y).Q'$

$R {=} \ldots z \ldots$

Assume that $z \notin \mathrm{fn}(P')$.

Depict the transition

$$(\boldsymbol{\nu} z)\,(\ P|R\ )\mid Q\ \rightarrow P' \mid (\boldsymbol{\nu} z)\,(\ R \mid [{}^{z}/_{y}]Q'\ )$$

as a flow graph (with scopes) and verify it using the reaction and congruence rules.

Example: Hand-Over Protocol

$$\begin{aligned}
\mathsf{Car}(\,\mathsf{talk}, \mathsf{swch}\,) &\stackrel{\text{def}}{=} \overline{\mathsf{talk}}.\mathsf{Car}\langle\,\mathsf{talk}, \mathsf{swch}\,\rangle \\
\mathsf{Base}_i &\stackrel{\text{def}}{=} \mathsf{talk}_i.\mathsf{Base}_i + \mathsf{give}_i(t', s').\overline{\mathsf{swch}_i}\langle t', s'\rangle.\mathsf{Idle}_i \\
\mathsf{Idle}_i &\stackrel{\text{def}}{=} \mathsf{alrt}_i().\mathsf{Base}_i \\
\mathsf{Ctre}_i &\stackrel{\text{def}}{=} \overline{\mathsf{give}_i}\langle\mathsf{talk}_{i\oplus 1}, \mathsf{swch}_{i\oplus 1}\rangle.\overline{\mathsf{alrt}_{i\oplus 1}}\langle\rangle.\mathsf{Ctre}_{i\oplus 1} \\
\mathsf{Syst}_i &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\cdots)\Big(\,\mathsf{Car}\langle\,\mathsf{talk}_1, \mathsf{swch}_1\,\rangle \mid \mathsf{Base}_1 \mid \mathsf{Idle}_2 \mid \mathsf{Ctre}_1\,\Big)
\end{aligned}$$

- ☐ Exercise: Observe that $\mathsf{Syst}_i \ (\xrightarrow{\tau})^3 \ \mathsf{Syst}_{i\oplus 1}$

Exercise: Overtaking Cars

A car $C\langle n, b, f \rangle$ on a road is connected to its back and front neighbor through b and f, respectively, while n just represents its identifier.

The road is assumed to be infinite, so we ignore any boundary problem, and it is static in the sense that no cars may enter or leave the road.

Define $C(x, b, f)$ such that a car may overtake another car. Beware of deadlocks and nested overtake attempts. You are not allowed to change the parameter x of instances of C.

$$\begin{aligned} \mathsf{Car}(x, b, f) &\stackrel{\text{def}}{=} \\ \mathsf{Fast}(x, b, f) &\stackrel{\text{def}}{=} \\ \mathsf{Slow}(x, b, f, b') &\stackrel{\text{def}}{=} \end{aligned}$$

LTS: Prefixes

actions $\mu \quad ::= \quad \tau \quad \mid \quad y\langle\vec{x}\rangle \quad \mid \quad \overline{y}\langle\vec{z}\rangle$

(TAU) $\tau.P \xrightarrow{\tau} P$

(OUT) $\overline{a}\langle\vec{b}\rangle.P \xrightarrow{\overline{a}\langle\vec{b}\rangle} P$

(INP) $$\frac{\vec{b} \subseteq \mathcal{N}}{a(\vec{x}).P \xrightarrow{a\vec{b}} [{}^{\vec{b}}/_{\vec{x}}]P} \text{ if } |\vec{b}| = |\vec{x}|$$

(COMM) $$\frac{P \xrightarrow{\overline{a}\langle\vec{b}\rangle} P' \qquad Q \xrightarrow{a\vec{b}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

LTS: Restriction

$$\text{(RES)}\ \frac{P \xrightarrow{\mu} P'}{(\boldsymbol{\nu} c)\, P \xrightarrow{\mu} (\boldsymbol{\nu} c)\, P'}\ \text{if } c \notin \mathrm{n}(\mu)$$

$$\text{(OPEN)}\ \frac{P \xrightarrow{(\boldsymbol{\nu}\vec{b})\, \overline{a}\langle\vec{z}\rangle} P'}{(\boldsymbol{\nu} c)\, P \xrightarrow{(\boldsymbol{\nu} c\vec{b})\, \overline{a}\langle\vec{z}\rangle} P'}\ \text{if } \vec{z} \ni c \notin \{a, \vec{b}\}$$

The label on transition $\xrightarrow{(\boldsymbol{\nu}\vec{b})\, \overline{a}\langle\vec{z}\rangle}$ is called *bound output*.
(Invariant: $\vec{b} \subset \vec{z} \land a \notin \vec{b}$.)

LTS: Parallel Composition

$$(\textsf{PAR})\ \frac{P \xrightarrow{\mu} P'}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q} \text{ if } \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$$

$$(\textsf{CLOSE})\ \frac{P \xrightarrow{a\,\vec{b}} P' \qquad Q \xrightarrow{(\boldsymbol{\nu}\vec{c})\,\overline{a}\langle\vec{b}\rangle} Q'}{P \,|\, Q \xrightarrow{\tau} (\boldsymbol{\nu}\vec{c})\,(P' \,|\, Q')} \text{ if } \{\vec{b}\} \cap \mathrm{fn}(P) = \emptyset$$

LTS: Miscellaneous

$$(\textsf{SUM})\ \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

$$(\textsf{REP})\ \frac{P \,|\, !\,P \xrightarrow{\mu} P'}{!\,P \xrightarrow{\mu} P'}$$

$$(\textsf{ALP})\ \frac{Q \xrightarrow{\mu} Q'}{P \xrightarrow{\mu} Q'}\ \text{if } P =_\alpha Q$$

Replication via Recursion

In the presence of process identifiers, ***recursion*** means that a process identifier A defined by

$$A(\,\vec{x}\,) \stackrel{\mathrm{def}}{=} Q_A$$

can be used in any process term P by means of instantiation.

$$Q_A = \cdots A\langle\,\vec{u}\,\rangle \cdots A\langle\,\vec{v}\,\rangle \cdots$$

Note that A could also be used like this within Q_A itself ...

$$P = \cdots A\langle\,\vec{y}\,\rangle \cdots A\langle\,\vec{z}\,\rangle \cdots$$

Using recursion, how can we model/simulate replication? Define a process identifier that, when instantiated, "behaves roughly like" $\boxed{\mathbf{!}\,P}$

Recursion via Replication

Using replication, recursion can be modeled through:

1. invent name a to stand for identifier A
2. for any R,
   let $\widehat{R}$ denote the result of replacing any call $A\langle\,\vec{w}\,\rangle$ by $\overline{a}\langle\vec{w}\rangle.\mathbf{0}$
3. replace P by

$$(\boldsymbol{\nu} a)\,(\ \widehat{P} \mid \mathbf{!}\, a(\vec{x}).\widehat{Q_A}\ )$$

Example:

$$\begin{array}{lcl} A(\,x_1, x_2\,) & \stackrel{\text{def}}{=} & \overline{x_1}\langle x_2\rangle.B\langle\,x_1, x_2\,\rangle \\ B(\,x_1, x_2\,) & \stackrel{\text{def}}{=} & \overline{x_2}\langle x_1\rangle.A\langle\,x_1, x_2\,\rangle \\ C(\,x_1, x_2\,) & \stackrel{\text{def}}{=} & \overline{x_1}\langle x_2\rangle.C\langle\,x_2, x_1\,\rangle \end{array}$$

Booleans

$$\begin{aligned}
\text{True}(\ l\ ) &\stackrel{\text{def}}{=} l(t,f).\bar{t}\langle\rangle \\
\text{False}(\ l\ ) &\stackrel{\text{def}}{=} l(t,f).\boxed{\qquad\qquad} \\
\text{If}(\ l, \text{foo}, \text{bar}\ ) &\stackrel{\text{def}}{=} (\boldsymbol{\nu} t f)\,\bar{l}\langle t,f\rangle.\big(\ \boxed{\qquad\qquad} + \boxed{\qquad\qquad}\ \big)
\end{aligned}$$

Check that for all P, Q:

$$(\boldsymbol{\nu} l)\left(\text{True}\langle\ l\ \rangle \mid \text{If}\langle\ l, \text{foo}, \text{bar}\ \rangle\right) \text{ “=” } \boxed{\qquad\qquad}$$

$$(\boldsymbol{\nu} l)\left(\text{False}\langle\ l\ \rangle \mid \text{If}\langle\ l, \text{foo}, \text{bar}\ \rangle\right) \text{ “=” } \boxed{\qquad\qquad}$$

Encoding Tuples

$$[\![\, \overline{y}\langle\vec{z}\rangle.P \,]\!] \quad \stackrel{\text{def}}{=}$$
$$[\![\, y(\vec{x}).P \,]\!] \quad \stackrel{\text{def}}{=}$$

Think about:

$$\overline{y}\langle z_1, z_2\rangle.P \mid y(x_1, x_2, x_3).Q$$
$$\longrightarrow$$
$$\overline{y}\langle z_1, z_2\rangle.P \mid y(x_1, x_2).Q \mid \overline{y}\langle w_1, w_2\rangle.R$$
$$\longrightarrow$$

$$[\![\, \overline{y}\langle\vec{z}\rangle.P \,]\!] \quad \stackrel{\text{def}}{=}$$
$$[\![\, y(\vec{x}).P \,]\!] \quad \stackrel{\text{def}}{=}$$

Asynchronous Pi Calculus

Synchronous communication means that a sender can only proceed with its continuation when a receiver has been found, and the handshake has taken place. Until then it is blocked.

Asynchronous communication means that a sender never waits, but goes ahead ... exactly like in distributed systems!

In the syntax of the calculus,
we just ***constrain the syntax of send prefixes to*** $\overline{a}\langle v\rangle.\mathbf{0}$.

For convenience, we may completely drop the send prefix and just use ***messages*** $\overline{a}\langle v\rangle$ instead, which can be added as another clause to the syntax productions for P.

Compare the "behaviors" of the following processes:

$$\overline{a}\langle v\rangle.P \qquad\qquad \overline{a}\langle v\rangle.\mathbf{0} \mid P \qquad\qquad \overline{a}\langle v\rangle \mid P$$

Encoding Synchrony

$$[\![P_1 \mid P_2]\!] \stackrel{\text{def}}{=} [\![P_1]\!] \mid [\![P_2]\!]$$

$$\vdots$$

$$[\![\overline{y}\langle z\rangle.P]\!] \stackrel{\text{def}}{=}$$

$$[\![y(x).P]\!] \stackrel{\text{def}}{=}$$

Encoding Summation

$$P \quad ::= \quad \ldots \quad \Big| \quad \sum_{i \in I} y_i(x).P_i \quad \Big| \quad \textsf{if } x \textsf{ then } P \textsf{ else } P$$

$$[\![\, \sum_{i \in I} y_i(x).P_i \,]\!] \quad \stackrel{\mathrm{def}}{=}$$

$$[\![\, y(x).P \,]\!] \quad \stackrel{\mathrm{def}}{=}$$

Encoding Lambda-Calculus

$$[\![\, x \,]\!](\, u \,) \quad \stackrel{\text{def}}{=} \quad \overline{x}\langle u\rangle$$

$$[\![\, \lambda x \, M \,]\!](\, u \,) \quad \stackrel{\text{def}}{=} \quad u(x, v).[\![\, M \,]\!]\langle\, v \,\rangle$$

$$\begin{aligned}[\![\, (M N) \,]\!](\, u \,) \quad \stackrel{\text{def}}{=} \quad (\boldsymbol{\nu} v) \;\Big(\quad & [\![\, M \,]\!]\langle\, v \,\rangle \\ \Big| \quad & (\boldsymbol{\nu} x)\, \overline{v}\langle x, u\rangle \\ \Big| \quad & \mathbf{!}\, x(u).[\![\, N \,]\!]\langle\, u \,\rangle \;\Big)\end{aligned}$$

Try to evaluate/encode $(\cdots((M_0 N_1) N_2) \cdots)$

“Final Words”

name-passing vs. **value-pasing**

- better pragmatics
- natural programming idioms
- semantic foundations for all (?) major programming styles
- ...
- security protocols via (s)pi-calculus
  ($\longrightarrow$ research at LAMP2 ...)