# Concurrency SS 2024
# Message Passing Concurrency

Peter Thiemann

July 3, 2024

# Outline

1. Message Passing
2. Go
3. Concurrent ML
4. Pi-Calculus

# Concurrency Flavors

## Shared Memory Concurrency

- processes interact by reading and writing shared variables
- locking etc. needed to demarcate critical regions

# Concurrency Flavors

## Shared Memory Concurrency

- processes interact by reading and writing shared variables
- locking etc. needed to demarcate critical regions

## Message Passing Concurrency

- processes interact by sending and receiving messages on shared communication channels

# Expressiveness

- message passing may be implemented using shared variables (viz. consumer/producer message queue implementations)
- shared variables may be implemented using message passing
  - model a reference by a thread and channels for reading and writing
  - reading on the “read” channel returns the current value
  - writing on the “write” channel spawns a new thread with the new value that manages the two channels from then on

# Synchronous vs. Asynchronous

- Receive operation blocks either way
- Given a channel with synchronous operations,
  - send asynchronously by sending in a spawned thread
- Given a channel with asynchronous operations.
  - establish a protocol to acknowledge receipts
  - pair each send operation with a receive for the acknowledgment

# First Incarnation

## Hoare's Communicating Sequential Processes (CSP)

**Prefix** $(x : B) \rightarrow P(x)$
await synchronizaton on event $x$ (an element of $B$) and then execute $P(x)$

**External Choice** $(a \rightarrow P \mid b \rightarrow Q)$
await synchronizaton on $a$ or $b$ and continue with $P$ or $Q$, respectively ($a \neq b$)

**Internal Choice** $(P \sqcap Q)$
continue nondeterministically with $P$ or $Q$

**Recursion** $\mu X \bullet P(X)$
process that recursively behaves like $P$

**Concurrency** $P \| Q$
$P$ runs in parallel with $Q$

**Sequential** (process local) variables, assignment, conditional, while

## Communication in CSP

- Special events
  - $c!v$ output $v$ on channel $c$
  - $c?x$ read from channel $c$ and bind to variable $x$
- Example: copy from channel *in* to channel *out*

$$COPY = \mu X \bullet (in?x \rightarrow (out!x) \rightarrow X)$$

- Example: generate sequence of ones

$$ONES = \mu X \bullet (in!1 \rightarrow X)$$

  Event $in!1$ synchronizes with $in?x$ and transmits the value to another process
- Example: last process behaves like /dev/null

$$ONES \| COPY \| \mu X \bullet (out?y \rightarrow X)$$

# CSP III

- CSP has influenced the design of numerous programming languages
  - Occam — programming “transputers”, processors with specific serial communication links
  - Golang — a programming language with cheap threads and channel based communication (Google 2011, `https://golang.org`)
  - CML — concurrent ML (John Reppy, 1999, `http://cml.cs.uchicago.edu/`)
- Golang and CML feature typed bidirectional channels
- Golang’s channels can be buffered

# Outline

```
// pi launches n goroutines to compute an
// approximation of pi.
func pi(n int) float64 {
        ch := make(chan float64)
        for k := 0; k <= n; k++ {
                go term(ch, float64(k))
        }
        f := 0.0
        for k := 0; k <= n; k++ {
                f += <-ch
        }
        return f
}

func term(ch chan float64, k float64) {
        ch <- 4 * math.Pow(-1, k) / (2*k + 1)
}
```

# Go II

## Example: Prime numbers

```
// Send the sequence 2, 3, 4, ... to channel 'ch'.
func Generate(ch chan<- int) {
  for i := 2; ; i++ {
    ch <- i // Send 'i' to channel 'ch'.
  }
}

// Copy from channel 'in' to channel 'out',
// removing values divisible by 'p'.
func Filter(in <-chan int, out chan<- int, p int) {
  for {
    i := <-in // Receive value from 'in'.
    if i%p != 0 {
      out <- i // Send 'i' to 'out'.
    }
  }
}
```

# Go IIa

## Example 'Prime numbers' continued

```
// The prime sieve: Daisy-chain Filter processes.
func main() {
  ch := make(chan int) // Create a new channel.
  go Generate(ch)      // Launch generator.
  for i := 0; i < 10; i++ {
    prime := <-ch
    fmt.Println(prime)
    ch1 := make(chan int)
    go Filter(ch, ch1, prime)
    ch = ch1
  }
}
```

# Outline

# Concurrent ML

- Synchronous message passing with first-class events
  - i.e., events are values in the language that can be passed as parameters and manipulated before they become part of a prefix
  - may be used to create new synchronization abstractions
- Originally for ML with implementations in Racket, Caml, Haskell, etc
- But ideas more widely applicable
- Requires threads to be very lightweight (i.e., thread creation at the cost of little more than a function call)

# CML's Channel Interface

```
type 'a channel (* messages passed on channels *)
val new_channel : unit -> 'a channel

type 'a event (* when sync'ed on, get an 'a *)
val send     : 'a channel -> 'a -> unit event
val receive  : 'a channel -> 'a event
val sync     : 'a event -> 'a
```

- `send` and `receive` return an event immediately
- `sync` blocks on the event until it happens
- This separation of concerns is important

# Simple Synchronous Operations

Define blocking send and receive operations:

```
let sendNow ch a = sync (send ch a)
let recvNow ch   = sync (receive ch)
```

- Each channel may have multiple senders and receivers that want to synchronize.
- Choice of pairing is nondeterministic, up to the implementation

```
type action = Put of float | Get of float
type account = action channel * float channel
let mkAcct () =
  let inCh  = new_channel() in
  let outCh = new_channel() in
  let bal   = ref 0.0       in (* state *)
  let rec loop () =
    (match recvNow inCh with (* blocks *)
      Put f -> bal := !bal +. f
    | Get f -> bal := !bal -. f); (* overdraw! *)
    sendNow outCh !bal; loop ()
in ignore(create loop ()); (* launch "server" *)
   (inCh,outCh) (* return channels *)
```

```
let mkAcct_functionally () =
  let inCh  = new_channel() in
  let outCh = new_channel() in
  let rec loop bal = (* state is loop-argument *)
    let newbal =
      match recvNow inCh with (* blocks *)
        Put f -> bal +. f
      | Get f -> bal -. f (* overdraw! *)
    in sendNow outCh newbal; loop newbal
  in ignore(create loop 0.0);
     (inCh,outCh)
```

- Viz. model a reference using channels

# Account Interface

Interface can abstract channels and concurrency from clients

```
type acct
val mkAcct : unit -> acct
val get : acct -> float -> float
val put : acct -> float -> float
```

- type `acct` is abstract, with `account` as possible implementation
- `mkAcct` creates a thread behind the scenes
- `get` and `put` make the server go round the loop once

Races are avoided by the implementation; the account server takes one request at a time

# Streams in CML

A stream is an infinite sequence of values produced lazily.

```
let nats = new_channel()
let rec loop i =
  sendNow nats i;
  loop (i+1)
let _ = create loop 0

let next_nat () = recvNow nats
```

# Introducing Choice

- `sendNow` and `recvNow` block until they find a communication partner (rendezvous).
- This behavior is not appropriate for many important synchronization patterns.
- Example:

  ```
  val add : int channel -> int channel -> int
  ```

  Should read the first value available on either channel to avoid blocking the sender.
- For this reason, `sync` is separate and there are further operators on events.

# Choose and Wrap

```
val choose : 'a event list -> 'a  event
val wrap   : 'a event -> ('a -> 'b) -> 'b event
val never  : 'a event
val always : 'a -> 'a event
```

- `choose`: creates an event that: when synchronized on, blocks until one of the events in the list happens
- `wrap`: the map function for channels; process the value returned by the event with a function (when it happens)
- `never = choose []`
- `always x`: synchronization is always possible; returns `x`
- further primitives omitted (e.g., timeouts)

# The Circuit Analogy

## Electrical engineer

- `send` and `receive` are ends of a gate
- `wrap` is logic attached to a gate
- `choose` is a multiplexer
- `sync` is getting a result

# The Circuit Analogy

## Electrical engineer

- `send` and `receive` are ends of a gate
- `wrap` is logic attached to a gate
- `choose` is a multiplexer
- `sync` is getting a result

## Computer scientist

- build data structure that describes a communication protocol
- first-class, so can be passed to `sync`
- events in interfaces so other libraries can compose

# Outline

# Pi-Calculus

- The Pi-Calculus is a low-level calculus meant to provide a formal foundation of computation by message passing.
- First presented in 1989 by Milner, Parrow, and Walker.
- Reference: Robin Milner's book "Communicating and Mobile Systems: the $\pi$-calculus", Cambridge University Press, 1999.
- Has given rise to a number of programming languages (Pict, JoCaml) and is acknowledged as a tool for business process modeling (BPML).
- Actively used and investigated in industry and academia.

# Pi-Calculus Features

**Primitives for describing and analysing global distributed infrastructure**

- process migration between peers
- process interaction via dynamic channels
- private channel communication.

# Pi-Calculus Features

## Primitives for describing and analysing global distributed infrastructure

- process migration between peers
- process interaction via dynamic channels
- private channel communication.

## Mobility

- processes move in the physical space of computing sites (successor: Ambient);
- processes move in the virtual space of linked processes;
- links move in the virtual space of linked processes (precursor: CCS, Calculus of Communicating Systems).

# Evolution from CCS

- CCS: synchronization on fixed events $a$

$$a.P \mid \overline{a}.Q \longrightarrow P \mid Q$$

- value-passing CCS

$$a(x).P \mid \overline{a}(v).Q \longrightarrow P\{x := v\} \mid Q$$

- Pi: synchronization on variable events (names) + name passing

$$x(y).P \mid \overline{x}(z).Q \longrightarrow P\{y := z\} \mid Q$$

# Example: Doctor's Surgery

## Based on example by Kramer and Eisenbach

A surgery consists of two doctors and one receptionist. Model the following interactions:

1. a patient checks in;
2. when a doctor is ready, the receptionist gives him the next patient;
3. the doctor gives prescription to the patient.

# Attempt Using CCS + Value Passing

1. Patient checks in with name and symptoms

$$P(n, s) = \overline{checkin}\langle n, s\rangle.?$$

2. Receptionist dispatches to next available doctor

$$R = checkin(n, s).(next_1.\overline{ans_1}\langle n, s\rangle.R + next_2.\overline{ans_2}\langle n, s\rangle.R)$$

3. Doctor gives prescription

$$D_i = \overline{next_i}.ans_i(n, s).?$$

- In CCS it's not possible to create an interaction between $P$ and $D_i$ because they don't have a shared channel name.

# Attempted Solution

Use patient's name as the name of a new channel.

$$D_i = \overline{next_i}.ans_i(n, s).\overline{n}\langle pre(s)\rangle.D_i$$

$$P(n, s) = \overline{checkin}\langle n, s\rangle.n(x).P'$$

Receptionist: Same code as before, but now the name of the channel is passed along.

$$R = checkin(n, s).(next_1.\overline{ans_1}\langle n, s\rangle.R + next_2.\overline{ans_2}\langle n, s\rangle.R)$$

# Improvement I

The doctor passes an answering channel to $R$.

$$D_i = \overline{next}(ans_i).ans_i(n, s).\overline{n}\langle pre(s)\rangle.D_i$$

$$R = checkin(n, s).next\langle ans\rangle.\overline{ans}\langle n, s\rangle.R)$$

WIth this encoding, the receptionist no longer depends on the number of doctors.
Patient: unchanged

$$P(n, s) = \overline{checkin}\langle n, s\rangle.n(x).P'$$

# Improvement II

- If two patients have the same name, then the current solution does not work.
- Solution: generate fresh channel names as needed
- Read $(\nu n)$ as “new n” (called restriction)

$$P(s) = (\nu n)\ \overline{checkin}\langle n, s\rangle.n(x).P'$$

- Same idea provides doctors with private identities
- Now same code for each doctor

$$D = (\nu a)\ \overline{next}(a).a(n, s).\overline{n}\langle pre(s)\rangle.D$$

- In $D \mid D \mid R$, every doctor creates fresh names

# Example: $n$-Place Buffer

Single buffer location (i.e., process)

$$B(in, out) = in(x).\overline{out}\langle x\rangle.B(in, out)$$

$n$-place buffer $B_n(i, o) =$

$$(\nu o_1)\ldots(\nu o_{n-1})(B(i, o_1) \mid \cdots \mid B(o_j, o_{j_1}) \mid \ldots B(o_{n-1}, o))$$

May still be done with CCS restriction $(\_\_) \setminus o_i$, which can close the scope of fixed names.

# Example: Unbounded Buffer

$$UB(in, out) = in(x).(\nu y)\ (UB(in, y) \mid B(x, y, out))$$

$$B(x, in, out) = \overline{out}\langle x\rangle.in(z).B(z, in, out)$$

- Drawback: Cells are never destroyed
- A elastic buffer, where cells are created and destroyed as needed, cannot be expressed in CCS.

# Formal Syntax of Pi-Calculus

## Identifiers

$$
\begin{array}{lcll}
u, v & ::= & a, b, c, \ldots & \text{names} \in \mathcal{N} \\
 & \mid & x, y, z, \ldots & \text{variables}
\end{array}
$$

# Formal Syntax of Pi-Calculus

## Identifiers

$$
\begin{array}{rcll}
u, v & ::= & a, b, c, \ldots & \text{names} \in \mathcal{N} \\
 & \mid & x, y, z, \ldots & \text{variables}
\end{array}
$$

## Pi-prefixes

$$
\begin{array}{rcll}
\pi & ::= & \overline{u}\langle \tilde{v} \rangle & \text{send list of names } \tilde{v} \text{ along channel } u \\
 & \mid & u(\tilde{y}) & \text{receive list of names } \tilde{y} \text{ along channel } u \\
 & \mid & \tau & \text{unobservable action}
\end{array}
$$

# Formal Syntax of Pi-Calculus

## Identifiers

$$
\begin{array}{rcll}
u, v & ::= & a, b, c, \ldots & \text{names} \in \mathcal{N} \\
 & \mid & x, y, z, \ldots & \text{variables}
\end{array}
$$

## Pi-prefixes

$$
\begin{array}{rcll}
\pi & ::= & \overline{u}\langle\tilde{v}\rangle & \text{send list of names } \tilde{v} \text{ along channel } u \\
 & \mid & u(\tilde{y}) & \text{receive list of names } \tilde{y} \text{ along channel } u \\
 & \mid & \tau & \text{unobservable action}
\end{array}
$$

## Pi-processes

$$
\begin{array}{rcll}
P & ::= & \sum_{i \in I} \pi_i.P_i & \text{summation over finite index set } I \\
 & \mid & P \mid Q & \text{parallel composition} \\
 & \mid & (\nu a)\ P & \text{restriction (binds a name)} \\
 & \mid & !P & \text{replication}
\end{array}
$$

# Summation (nondeterministic guarded choice)

- In $\sum_{i \in I} \pi_i.P_i$, the process $P_i$ is guarded by the action $\pi_i$
- 0 stands for the empty sum (i.e., $I = \emptyset$)
- $\pi.P$ abbreviates a singleton sum
- The output process $\overline{u}\langle\tilde{v}\rangle.P$
  sends the list of free names $\tilde{v}$ over $u$ and continues as $P$
- The input process $u(\tilde{z})..P$
  binds the list of distinct names $\tilde{z}$. It can receive any names $\tilde{v}$ over $x$ and continues as $P\{\tilde{z} := \tilde{v}\}$

# Summation (nondeterministic guarded choice)

- In $\sum_{i \in I} \pi_i . P_i$, the process $P_i$ is guarded by the action $\pi_i$
- 0 stands for the empty sum (i.e., $I = \emptyset$)
- $\pi . P$ abbreviates a singleton sum
- The output process $\overline{u}\langle \tilde{v} \rangle . P$
  sends the list of free names $\tilde{v}$ over $u$ and continues as $P$
- The input process $u(\tilde{z})..P$
  binds the list of distinct names $\tilde{z}$. It can receive any names $\tilde{v}$ over $x$ and continues as $P\{\tilde{z} := \tilde{v}\}$

**Examples**

$$x(z).\overline{y}\langle z \rangle \qquad x(z).\overline{z}\langle y \rangle \qquad x(z).\overline{z}\langle y \rangle + \overline{w}\langle v \rangle$$

# Restriction

- The restriction $(\nu a)\ P$ binds a fresh name $a$ in $P$.
- Processes in $P$ can use $a$ to act among each others.
- $a$ is not visible outside the restriction.

# Restriction

- The restriction $(\nu a)\ P$ binds a fresh name $a$ in $P$.
- Processes in $P$ can use $a$ to act among each others.
- $a$ is not visible outside the restriction.

## Example

$$(\nu a)\ ((a(z).\overline{z}\langle y\rangle + \overline{w}\langle v\rangle) \mid \overline{a}\langle u\rangle)$$

# Replication

- The replication $!P$ can be regarded as a process consisting of arbitrary many compositions of $P$.
- As an equation: $!P = P \mid !P$.

# Replication

- The replication $!P$ can be regarded as a process consisting of arbitrary many compositions of $P$.
- As an equation: $!P = P \mid !P$.

## Examples

- $!x(z).\overline{y}\langle z\rangle.0$
  Repeatedly receive a name over $x$ and send it over $y$.
- $!x(z).!\overline{y}\langle z\rangle.0$
  Repeatedly receive a name over $x$ and repeatedly send it over $y$.

# Free variables and free names

| $P$ | $fv(P)$ | $fn(P)$ | |
|---|---|---|---|
| $x$ | $\{x\}$ | $\{\}$ | $*$ |
| $a$ | $\{\}$ | $\{a\}$ | $*$ |
| $0$ | $\{\}$ | $\{\}$ | |
| $P \mid Q$ | $fv(P) \cup fv(Q)$ | $fn(P) \cup fn(Q)$ | |
| $(\nu a)P$ | $fv(P)$ | $fn(P) \setminus \{a\}$ | $*$ |
| $!P$ | $fv(P)$ | $fn(P)$ | |
| $\overline{u}\langle\tilde{v}\rangle.P$ | $fv(u) \cup fv(\tilde{v}) \cup fv(P)$ | $fn(u) \cup fn(\tilde{v}) \cup fn(P)$ | |
| $u(\tilde{z}).P$ | $fv(u) \cup (fv(P) \setminus \{\tilde{z}\})$ | $fn(u) \cup fn(P)$ | $*$ |

- Both $u(-)$ and $(\nu-)$ are binders.
- A term is closed if it has no free variables (otherwise open).
- Consider $P = (\nu b)a(x).(\overline{x}\langle z\rangle.0 \mid \overline{x}\langle b\rangle.0)$.
  It highlights $fn(P) = \{a\}$ and $fv(P) = \{z\}$.

# $\alpha$-conversion

$\alpha$-conversion (written $P =_\alpha Q$) is the consistent renaming of bound variables or bound names. It must not change or hide free variables/names.

- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) =_\alpha (\nu d)(\overline{d}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle d\rangle.0)$

# $\alpha$-conversion

$\alpha$-conversion (written $P =_\alpha Q$) is the consistent renaming of bound variables or bound names. It must not change or hide free variables/names.

- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) =_\alpha (\nu d)(\overline{d}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle d\rangle.0)$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq_\alpha (\nu b)(\overline{b}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle b\rangle.0)$
  $b \in fn(lhs)$, but $b \notin fn(rhs)$

# $\alpha$-conversion

$\alpha$-conversion (written $P =_\alpha Q$) is the consistent renaming of bound variables or bound names. It must not change or hide free variables/names.

- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) =_\alpha (\nu d)(\overline{d}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle d\rangle.0)$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq_\alpha (\nu b)(\overline{b}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle b\rangle.0)$
  $b \in fn(lhs)$, but $b \notin fn(rhs)$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq_\alpha (\nu c)(\overline{c}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle c\rangle.0)$
  $fn((\nu c)\overline{c}\langle a\rangle.0) = \{a\}$, but $fn((\nu c)\overline{c}\langle c\rangle.0) = \{\}$

# $\alpha$-conversion

$\alpha$-conversion (written $P =_\alpha Q$) is the consistent renaming of bound variables or bound names. It must not change or hide free variables/names.

- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) =_\alpha (\nu d)(\overline{d}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle d\rangle.0)$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq_\alpha (\nu b)(\overline{b}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle b\rangle.0)$
  $b \in fn(lhs)$, but $b \notin fn(rhs)$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq_\alpha (\nu c)(\overline{c}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle c\rangle.0)$
  $fn((\nu c)\overline{c}\langle a\rangle.0) = \{a\}$, but $fn((\nu c)\overline{c}\langle c\rangle.0) = \{\}$
- $(\nu a)(\overline{a}\langle b\rangle.0 \mid (\nu c)\overline{c}\langle a\rangle.0) \neq (\nu c)(\overline{c}\langle b\rangle.0 \mid (\nu e)\overline{e}\langle c\rangle.0)$
  after $\alpha$-converting the subprocess

# Substitution

A substitution $[x := a]$ applied to a process $P$ (as in $P[x := a]$) replaces all free occurrences of variable $x$ by name $a$.
Substitution is capture-avoiding, that is, it $\alpha$-converts bound names as needed.
Example:

$$
\begin{aligned}
& ((\nu d)(\overline{a}\langle b\rangle.0 \mid \overline{a}\langle d\rangle.0 \mid \overline{a}\langle x\rangle.0))[x := d] \\
= \; & ((\nu e)(\overline{a}\langle b\rangle.0 \mid \overline{a}\langle e\rangle.0 \mid \overline{a}\langle d\rangle.0))
\end{aligned}
$$

# Variation: Monadic Pi-Calculus

Send and receive primitives are restricted to pass single names.

## Monadic pi-prefixes

| | | | |
|---|---|---|---|
| $\pi$ | $::=$ | $\overline{u}\langle v\rangle$ | send name $v$ along $u$ |
| | $\mid$ | $u(y).$ | receive name $y$ along $u$ |
| | $\mid$ | $\tau$ | unobservable action |

Monadic processes defined as before on top of monadic pi-actions.

# Simulating Pi with Monadic Pi

## First attempt

- Obvious idea for a translation from Pi to monadic Pi:

$$
\begin{array}{rcl}
\overline{u}\langle\tilde{v}\rangle & \to & \overline{u}\langle v_1\rangle \ldots \overline{u}\langle v_n\rangle \\
u(\tilde{y}) & \to & u(y_1) \ldots u(y_n)
\end{array}
$$

# Simulating Pi with Monadic Pi

## First attempt

- Obvious idea for a translation from Pi to monadic Pi:

$$\begin{array}{rcl} \overline{u}\langle\tilde{v}\rangle & \rightarrow & \overline{u}\langle v_1\rangle \ldots \overline{u}\langle v_n\rangle \\ u(\tilde{y}) & \rightarrow & u(y_1)\ldots u(y_n) \end{array}$$

- Does not work

# Simulating Pi with Monadic Pi

## First attempt

- Obvious idea for a translation from Pi to monadic Pi:

$$\begin{array}{rcl} \overline{u}\langle\tilde{v}\rangle & \rightarrow & \overline{u}\langle v_1\rangle\ldots\overline{u}\langle v_n\rangle \\ u(\tilde{y}) & \rightarrow & u(y_1)\ldots u(y_n) \end{array}$$

- Does not work
- Counterexample

$$x(y_1, y_2).P \mid \overline{x}\langle z_1, z_2\rangle.Q \mid \overline{x}\langle z_1', z_2'\rangle.Q'$$

# Simulating Pi with Monadic Pi

## Correct encoding

Suppose that $w \notin fn(P, Q)$

$$\begin{array}{lcl} \overline{x}\langle\tilde{y}\rangle.P & \rightarrow & (\nu w)\,\overline{x}\langle w\rangle\overline{w}\langle y_1\rangle \ldots \overline{w}\langle y_n\rangle.P^{\dagger} \\ x(\tilde{y}).Q & \rightarrow & x(w).w(y_1)\ldots w(y_n).Q^{\dagger} \end{array}$$

where $P^{\dagger}$ and $Q^{\dagger}$ are recursively transformed in the same way.

# Recursion by Replication

The Pi-calculus can encode recursion. Suppose a process is defined using recursion

$$A(\tilde{x}) = Q_A$$

where $Q_A$ contains calls to $A$ and process $P$ is the scope of $A$. The translation is given by

1. introduce a new name $a$ to stand for $A$;
2. for any process $R$, write $\hat{R}$ for the result of replacing every call $A(\tilde{w})$ by $\overline{a}\langle\tilde{w}\rangle$;
3. replace $P$ and the old definition of $A$ by

$$\hat{\hat{P}} = (\nu a)\ (\hat{P}\ |!a(\tilde{x}).\hat{Q}_A)$$

# Structural Congruence

The reduction semantics of the $\pi$-calculus is inspired by the Chemical Abstract Machine (CHAM) of Berry and Boudol. Processes “float around” like molecules in a solution using structural congruence ($\equiv$) and “react” using a reduction relation ($\longrightarrow$).

The intuition is that if $P \equiv Q$ then we consider $P$ and $Q$ completely interchangeable.

Structural congruence is an equivalence relation, it is preserved by all the syntactic operators, and it contains $\alpha$-equivalence (renaming of bound names).

# Structural Congruence (Axioms)

| | |
|---|---|
| $P \equiv P$ | reflexivity |
| $P \equiv Q \Rightarrow Q \equiv P$ | symmetry |
| $P \equiv R$ and $R \equiv Q \Rightarrow P \equiv Q$ | transitivity |
| $P \equiv Q \Rightarrow (\nu a)P \equiv (\nu a)Q$ | congruence-res |
| $P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$ | congruence-par |
| $P \equiv Q \Rightarrow \pi.P \Rightarrow \pi.Q$ | congruence-comm |
| $P \equiv Q \Rightarrow !P \equiv !Q$ | congruence-repl |
| $P =_\alpha Q \Rightarrow P \equiv Q$ | |

# Structural Congruence (pi-calculus specific)

Structural congruence $\equiv$ is the smallest congruence on terms $P$ of the monadic pi-calculus

1. $P + 0 \equiv P, \quad P + Q \equiv Q + P, \quad P + (Q + R) \equiv (P + Q) + R$
2. $P \mid 0 \equiv P, \quad P \mid Q \equiv Q \mid P, \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
3. $(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$ if $a \notin \mathrm{fn}(P)$, $\quad (\nu a)0 \equiv 0$, $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$
4. $!P \equiv P \mid !P$

- Structural congruence relates processes that should behave the same.
- It simplifies the definition of the semantics because we can assume that processes that “react” with each other are side by side.

# Reduction Semantics of Pi

The reduction relation is the smallest binary relation $\longrightarrow$ on terms satisfying

TAU

$$\tau.P \longrightarrow P$$

STRUCT

$$\frac{P' \equiv P \qquad P \longrightarrow Q \qquad Q \equiv Q'}{P' \longrightarrow Q'}$$

REACT

$$(\overline{a}\langle v\rangle.P_1 + Q) \mid (a(z).P_2 + R) \longrightarrow P_1 \mid P_2[z := v]$$

PAR

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

RES

$$\frac{P \longrightarrow P'}{(\nu a)P \longrightarrow (\nu a)P'}$$

# Examples

## Infinite behavior

$$
\begin{array}{ll}
 & \overline{a}\langle b\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0 \\
\equiv & \overline{a}\langle b\rangle.0 \mid a(x).\overline{a}\langle x\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0 \\
\longrightarrow & \overline{a}\langle b\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0
\end{array}
$$

# Examples

## Infinite behavior

$$\begin{array}{rl} & \overline{a}\langle b\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0 \\ \equiv & \overline{a}\langle b\rangle.0 \mid a(x).\overline{a}\langle x\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0 \\ \longrightarrow & \overline{a}\langle b\rangle.0 \mid !a(x).\overline{a}\langle x\rangle.0 \end{array}$$

## Nondeterminism

$$\overline{a}\langle b\rangle.0 \mid \overline{a}\langle d\rangle.0 \mid a(x).\overline{c}\langle x\rangle.0 \quad \begin{array}{l} \nearrow \ \overline{a}\langle d\rangle.0 \mid \overline{c}\langle b\rangle.0 \\ \searrow \ \overline{a}\langle b\rangle.0 \mid \overline{c}\langle d\rangle.0 \end{array}$$

# Example (Mobility - Scope extrusion)

$$\mathcal{Q} = (\nu a)(\overline{b}\langle a\rangle.P \mid R) \mid b(y).Q$$

where $a \notin \text{fn}(P) \cup \text{fn}(Q)$.

We have $\mathcal{Q} \longrightarrow P \mid (\nu a)(R \mid Q[y := a])$ because

1. $(\overline{b}\langle a\rangle.P) \mid (b(y).Q) \longrightarrow P \mid Q[y := a]$ due to [react] and [struct]
2. $(\overline{b}\langle a\rangle.P) \mid (b(y).Q) \mid R \longrightarrow P \mid Q[y := a] \mid R$ due to item 1 and [par]
3. $(\overline{b}\langle a\rangle.P) \mid R \mid (b(y).Q) \longrightarrow P \mid R \mid Q[y := a]$ due to item 2 and [struct]
4. $(\nu a)((\overline{b}\langle a\rangle.P) \mid R \mid (b(y).Q)) \longrightarrow (\nu a)(P \mid R \mid Q[y := a])$ due to item 3 and [res]
5. $(\nu a)(\overline{b}\langle a\rangle.P \mid R) \mid (b(y).Q) \longrightarrow P \mid (\nu a)(R \mid Q[y := a])$ due to item 4 and [struct]

# Small Agents (1)

**Forwarder $FW(a, b) = a(z).\overline{b}\langle z \rangle.0$**

Forwards messages on channel $a$ to channel $b$

$$FW(a, b) \mid \overline{a}\langle d \rangle.0 \longrightarrow \overline{b}\langle d \rangle.0$$

$$(\nu b)(FW(a, b) \mid FW(b, c))\overline{a}\langle d \rangle.0 \longrightarrow^* \overline{c}\langle d \rangle.0$$

# Small Agents (1)

## Forwarder $FW(a, b) = a(z).\overline{b}\langle z\rangle.0$

Forwards messages on channel $a$ to channel $b$

$$FW(a, b) \mid \overline{a}\langle d\rangle.0 \longrightarrow \overline{b}\langle d\rangle.0$$

$$(\nu b)(FW(a, b) \mid FW(b, c))\overline{a}\langle d\rangle.0 \longrightarrow^{*} \overline{c}\langle d\rangle.0$$

## Duplicator $D(a, b, c) = a(x).(\overline{b}\langle x\rangle.0 \mid \overline{c}\langle x\rangle.0))$

Duplicates messages on $a$ to channels $b$ and $c$

$$D(a, b, c) \mid \overline{a}\langle d\rangle.0 \longrightarrow \overline{b}\langle d\rangle.0 \mid \overline{c}\langle d\rangle.0$$

$$(\nu b)(D(a, b, c_1) \mid D(b, c_1, c_2)) \mid \overline{a}\langle d\rangle.0$$
$$\longrightarrow \overline{c_1}\langle d\rangle.0 \mid \overline{c_2}\langle d\rangle.0 \mid \overline{c_3}\langle d\rangle.0$$

# Small Agents (1)

**Forwarder $FW(a, b) = a(z).\overline{b}\langle z\rangle.0$**

Forwards messages on channel $a$ to channel $b$

$$FW(a, b) \mid \overline{a}\langle d\rangle.0 \longrightarrow \overline{b}\langle d\rangle.0$$

$$(\nu b)(FW(a, b) \mid FW(b, c))\overline{a}\langle d\rangle.0 \longrightarrow^* \overline{c}\langle d\rangle.0$$

**Duplicator $D(a, b, c) = a(x).(\overline{b}\langle x\rangle.0 \mid \overline{c}\langle x\rangle.0))$**

Duplicates messages on $a$ to channels $b$ and $c$

$$D(a, b, c) \mid \overline{a}\langle d\rangle.0 \longrightarrow \overline{b}\langle d\rangle.0 \mid \overline{c}\langle d\rangle.0$$

$$(\nu b)(D(a, b, c_1) \mid D(b, c_1, c_2)) \mid \overline{a}\langle d\rangle.0$$
$$\longrightarrow \overline{c_1}\langle d\rangle.0 \mid \overline{c_2}\langle d\rangle.0 \mid \overline{c_3}\langle d\rangle.0$$

**Killer $K(a) = a(z).0$**

Kills a message on $a$

# Small Agents in action

- $a(z).(P \mid Q)$ can be expressed by ...
- $(\nu c_1, c_2)(D(a, c_1, c_2) \mid c_1(z).P \mid c_2(z).Q)$
- (in what sense do these processes behave the same?)
- Example: $a(z).(\overline{b}\langle z\rangle.0 \mid 0)$
- behaves like $(\nu c_1, c_2)(D(a, c_1, c_2) \mid FW(c_1, b) \mid K(c_2))$

# Small Agents (2)

## Identity Receptor $I(a) = !FW(a, a)$

Forwards messages for $a$ on $a$

$$\overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \ldots$$

# Small Agents (2)

## Identity Receptor $I(a) = !FW(a, a)$

Forwards messages for $a$ on $a$

$$\overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \ldots$$

## Equator $EQ(a, b) = !FW(a, b) \mid !FW(b, a)$

Forwards all messages for $a$ to $b$ and vice versa, which makes $a$ and $b$ receptive to messages on either channel.

$$\overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \overline{b}\langle d\rangle.0 \mid EQ(a, B) \longrightarrow \overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \ldots$$

# Small Agents (2)

## Identity Receptor $I(a) = !FW(a, a)$

Forwards messages for $a$ on $a$

$$\overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \ldots$$

## Equator $EQ(a, b) = !FW(a, b) \mid !FW(b, a)$

Forwards all messages for $a$ to $b$ and vice versa, which makes $a$ and $b$ receptive to messages on either channel.

$$\overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \overline{b}\langle d\rangle.0 \mid EQ(a, B) \longrightarrow \overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \ldots$$

## Omega $\Omega = (\nu a)(!FW(a, a) \mid \overline{a}\langle a\rangle.0)$

Reduces infinitely to itself.

$$\Omega = (\nu a)(!(a(z).\overline{a}\langle z\rangle.0) \mid \overline{a}\langle a\rangle.0) \longrightarrow \Omega \longrightarrow \ldots$$

# Small Agents (2)

## Identity Receptor $I(a) = !FW(a, a)$

Forwards messages for $a$ on $a$

$$\overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \overline{a}\langle d\rangle.0 \mid I(a) \longrightarrow \dots$$

## Equator $EQ(a, b) = !FW(a, b) \mid !FW(b, a)$

Forwards all messages for $a$ to $b$ and vice versa, which makes $a$ and $b$ receptive to messages on either channel.

$$\overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \overline{b}\langle d\rangle.0 \mid EQ(a, B) \longrightarrow \overline{a}\langle d\rangle.0 \mid EQ(a, b) \longrightarrow \dots$$

## Omega $\Omega = (\nu a)(!FW(a, a) \mid \overline{a}\langle a\rangle.0)$

Reduces infinitely to itself.

$$\Omega = (\nu a)(!(a(z).\overline{a}\langle z\rangle.0) \mid \overline{a}\langle a\rangle.0) \longrightarrow \Omega \longrightarrow \dots$$

## New Name Generator $NN(a) = !a(z).(\nu b)\overline{z}\langle b\rangle.0$

$$\begin{aligned}\overline{a}\langle c\rangle.0 \mid \overline{a}\langle d\rangle.0 \mid NN(a) \quad &\longrightarrow (\nu b)\overline{c}\langle b\rangle.0 \mid \overline{a}\langle d\rangle.0 \mid NN(a) \\ &\longrightarrow (\nu b)\overline{c}\langle b\rangle.0 \mid (\nu b')\overline{d}\langle b'\rangle.0 \mid NN(a)\end{aligned}$$

# Example (Mobility and name generation)

## Internet connection

Client and server connect via dynamically assigned ports

$$\begin{aligned}\textbf{Client}(a) &= (\nu c)(\overline{a}\langle c\rangle.0 \mid c(x).\textbf{Client}_1(c,x))\\ \textbf{Server}(a) &= a(y).(\nu s)(\overline{y}\langle s\rangle.0 \mid \textbf{Server}_1(y,s))\end{aligned}$$

The names $c$ and $s$ are local to client and server.

$$\begin{aligned}& \textbf{Client}(a) \mid \textbf{Server}(a)\\ \longrightarrow\ & (\nu c)(c(x).\textbf{Client}_1(c,x) \mid (\nu s)(\overline{c}\langle s\rangle.0 \mid \textbf{Server}_1(c,s)))\\ \longrightarrow\ & (\nu c)(\nu s)(\textbf{Client}_1(c,s) \mid \textbf{Server}_1(c,s))\end{aligned}$$

Realistically, the server should be able to connect to multiple clients. So we'd represent it by $!\textbf{Server}(a)$.

# Towards a semantics with a transition system

**Motto**

Semantics without congruence

**Free Output:** represented by $\alpha = \overline{a}b$, where $a$ is the <u>subject</u> of $\alpha$, $b$ its <u>object</u>
$\mathrm{fn}(\alpha) = \{a, b\}$ and $\mathrm{bn}(\alpha) = \{\}$.

**Input:** $\alpha = ab$ with subject $a$, object $b$
$\mathrm{fn}(\alpha) = \{a, b\}$ and $\mathrm{bn}(\alpha) = \{\}$.

**Bound Output:** $\alpha = \overline{a}(b)$ with subject $a$, object $c$
$\mathrm{fn}(\alpha) = \{a\}$ and $\mathrm{bn}(\alpha) = \{c\}$.

# Towards a semantics with a transition system

## Motto

Semantics without congruence

## Pi-calculus actions

$$\alpha \quad ::= \quad \overline{a}b \mid ab \mid \overline{a}(y) \mid \tau$$

Free Output: represented by $\alpha = \overline{a}b$, where $a$ is the <u>subject</u> of $\alpha$, $b$ its <u>object</u>
$\mathrm{fn}(\alpha) = \{a, b\}$ and $\mathrm{bn}(\alpha) = \{\}$.

Input: $\alpha = ab$ with subject $a$, object $b$
$\mathrm{fn}(\alpha) = \{a, b\}$ and $\mathrm{bn}(\alpha) = \{\}$.

Bound Output: $\alpha = \overline{a}(b)$ with subject $a$, object $c$
$\mathrm{fn}(\alpha) = \{a\}$ and $\mathrm{bn}(\alpha) = \{c\}$.

# LTS Semantics of Pi

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

IN
$$a(z).P \xrightarrow{ab} P[z := b]$$

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

SUM-L
$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P' + Q}$$

SUM-R
$$\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} P + Q'}$$

PAR-L
$$\frac{P \xrightarrow{\alpha} P' \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

PAR-R
$$\frac{Q \xrightarrow{\alpha} Q' \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

REACT-L
$$\frac{P \xrightarrow{\overline{a}b} P' \qquad Q \xrightarrow{ab} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

REACT-R
$$\frac{P \xrightarrow{ab} P' \qquad Q \xrightarrow{\overline{a}b} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

# LTS Semantics of Pi (part 2)

RES

$$\frac{P \xrightarrow{\alpha} P' \qquad a \notin n(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'}$$

OPEN

$$\frac{P \xrightarrow{\overline{a}c} P' \qquad a \neq c}{(\nu c)P \xrightarrow{\overline{a}(c)} P'}$$

CLOSE-L

$$\frac{P \xrightarrow{\overline{a}(c)} P' \qquad Q \xrightarrow{ac} Q' \qquad c \notin \mathrm{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu c)(P' \mid Q')}$$

CLOSE-R

...

REP-ACT

$$\frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

REP-COMM

$$\frac{P \xrightarrow{\overline{a}b} P' \qquad P \xrightarrow{ab} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

REP-CLOSE

$$\frac{P \xrightarrow{\overline{a}(c)} P' \qquad P \xrightarrow{ac} P'' \qquad c \notin \mathrm{fn}(P)}{!P \xrightarrow{\tau} (\nu c)(P' \mid P'') \mid !P}$$

# Properties

## Harmony Lemma

1. $P \equiv\xrightarrow{\alpha} P'$ implies $P \xrightarrow{\alpha}\equiv P'$
2. $P \longrightarrow P'$ if and only if $P \xrightarrow{\tau}\equiv P'$

# Wrapup

- The pi-calculus is a foundational calculus for concurrency.
- It is regarded as the concurrency counterpart for the lambda calculus.
- Lambda calculus can be encoded in pi-calculus.