Concurrency WS 2014/15
Message Passing Concurrency

Peter Thiemann

February 3, 2015
Outline

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

Shared Memory Concurrency
- Processes interact by reading and writing shared variables
- Locking etc. needed to demarcate critical regions
Concurrent 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

<table>
<thead>
<tr>
<th><strong>Hoare’s Communicating Sequential Processes (CSP)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prefix</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Choice</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Recursion</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Sequential</strong></td>
</tr>
</tbody>
</table>
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`

  \[
  \text{COPY} = \mu X \cdot (\text{in}?x \rightarrow (\text{out}!x) \rightarrow X)
  \]

- **Example:** generate sequence of ones

  \[
  \text{ONES} = \mu X \cdot (\text{in}!1 \rightarrow X)
  \]

  Event `in!1` synchronizes with `in?x` and transmits the value to the other process

- **Example:** last process behaves like `/dev/null`

  \[
  \text{ONES} \parallel \text{COPY} \parallel \mu X \cdot (\text{out}?y \rightarrow X)
  \]
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)
- Golang and CML feature typed bidirectional channels
- Golang’s channels are synchronous
Outline

1. Message Passing
2. Go
3. Concurrent ML
4. Pi-Calculus
// 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)
}
// 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'.
        }
    }
}
// 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
}
}
1. Message Passing
2. Go
3. Concurrent ML
4. Pi-Calculus
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

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
Define blocking send and receive operations:

```ocaml
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

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
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
A stream is an infinite sequence of values produced lazily.

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

1. Message Passing
2. Go
3. Concurrent ML
4. Pi-Calculus
The Pi-Calculus is a low-level calculus meant to provide a formal foundation of computation by message passing. Due to Robin Milner (see book “Communicating and Mobile Systems”, 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).
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 (predecessor: CCS).
Evolution from CCS

- CCS: synchronization on fixed events
  
  \[ a.P \ | \ \overline{a}.Q \rightarrow P \ | \ Q \]

- value-passing CCS
  
  \[ a(x).P \ | \ \overline{a}(v).Q \rightarrow P\{x := v\} \ | \ Q \]

- Pi: synchronization on variable events (names) + value passing
  
  \[ x(y).P \ | \ \overline{x}(z).Q \rightarrow P\{y := z\} \ | \ Q \]
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

\[ P(n, s) = \text{checkin}(n, s).? \]

2. Receptionist dispatches to next available doctor

\[ R = \text{checkin}(n, s). (\text{next}_1.\text{ans}_1(n, s).R + \text{next}_2.\text{ans}_2(n, s).R) \]

3. Doctor gives prescription

\[ D_i = \text{next}_i.\text{ans}_i(n, s).? \]

- In CCS it’s not possible to create an interaction between \( P \) and \( D_i \) because they don’t have a shared channel name.
Use patient’s name as the name of a new channel.

\[ D_i = \text{next}_i.\text{ans}_i(n, s).\overline{n}\langle \text{pre}(s) \rangle.D_i \]

\[ P(n, s) = \text{checkin}\langle n, s \rangle.n(x).P' \]

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

\[ R = \text{checkin}(n, s).(\text{next}_1.\overline{\text{ans}_1}\langle n, s \rangle.R + \text{next}_2.\overline{\text{ans}_2}\langle n, s \rangle.R) \]
The doctor passes an answering channel to $R$.

$$D_i = \text{next}(\text{ans}_i) \cdot \text{ans}_i(n, s) \cdot \overline{n}\langle\text{pre}(s)\rangle \cdot D_i$$

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

With this encoding, the receptionist no longer depends on the number of doctors.

Patient: unchanged

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

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

Same idea provides doctors with private identities

Now same code for each doctor

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

In $D | D | R$, every doctor creates fresh names
Example: \( n \)-Place Buffer

Single buffer location

\[
B(in, out) = \text{in}(x).\overline{\text{out}}\langle x\rangle.B(in, out)
\]

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

\[
(\nu o_1) \ldots (\nu o_{n-1})(B(i, o_1) \mid \ldots \mid B(o_j, o_j) \mid \ldots B(o_{n-1}, o))
\]

May still be done with CCS restriction (___) \( \backslash o_i \), which can close the scope of fixed names.
Example: Unbounded Buffer

\[ UB(in, out) = in(x).(\nu y) (UB(in, y) | B(x, y, out)) \]
\[ B(in, out) = in(x).B(x, in, out) \]
\[ B(x, in, out) = \overline{out}(x).B(in, out) \]

- 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

Let $x, y, z, \ldots$ range over an infinite set $\mathcal{N}$ of names.

**Pi-actions**

$$\pi ::= x\langle \tilde{y} \rangle \quad \text{send list of names } \tilde{y} \text{ along channel } x$$  
$$\quad | x(\tilde{y}) \quad \text{receive list of names } \tilde{y} \text{ along channel } x$$  
$$\quad | \tau \quad \text{unobservable action}$$
Let $x, y, z, \ldots$ range over an infinite set $\mathcal{N}$ of names.

**Pi-actions**

\[
\pi ::= \bar{x}\langle\tilde{y}\rangle \quad \text{send list of names } \tilde{y} \text{ along channel } x \\
\quad | \quad x(\tilde{y}) \quad \text{receive list of names } \tilde{y} \text{ along channel } x \\
\quad | \quad \tau \quad \text{unobservable action}
\]

**Pi-processes**

\[
P ::= \sum_{i \in I} \pi_i. P_i \quad \text{summation over finite index set } I \\
\quad | \quad P \mid Q \quad \text{parallel composition} \\
\quad | \quad (\nu x) \ P \quad \text{restriction} \\
\quad | \quad ! P \quad \text{replication}
\]
In $\sum_{i \in I} \pi_i \cdot P_i$, the process $P_i$ is guarded by the action $\pi_i$.

0 stands for the empty sum (i.e., $I = \emptyset$).

$\pi \cdot P$ abbreviates a singleton sum.

The output process $x\langle \tilde{y} \rangle \cdot P$ can send the list of free names $\tilde{y}$ over $x$ and continue as $P$.

The input process $x(\tilde{z}). P$ binds the list distinct names $\tilde{z}$. It can receive any names $\tilde{u}$ over $x$ and continues as $P\{\tilde{z} := \tilde{u}\}$.
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 $\overline{x} \langle \tilde{y} \rangle . P$ can send the list of free names $\tilde{y}$ over $x$ and continue as $P$.

The input process $x(\tilde{z}) . P$ binds the list distinct names $\tilde{z}$. It can receive any names $\tilde{u}$ over $x$ and continues as $P\{\tilde{z} := \tilde{u}\}$.

**Examples**

- $x(z) . \overline{y} \langle z \rangle$
- $x(z) . \overline{z} \langle y \rangle$
- $x(z) . \overline{z} \langle y \rangle + \overline{w} \langle v \rangle$
- The restriction \( (\nu z) P \) binds \( z \) in \( P \).
- Processes in \( P \) can use \( z \) to act among each other.
- \( z \) is not visible outside the restriction.
The restriction $((\nu z) P)$ binds $z$ in $P$.

Processes in $P$ can use $z$ to act among each other.

$z$ is not visible outside the restriction.

Example

$$(\nu x) \left( ((x(z).\overline{z}\langle y \rangle + \overline{w}\langle v \rangle) \mid \overline{x}\langle u \rangle) \right)$$
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: $!P = P | !P$.

**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 $y$. 
Variation: Monadic Pi-Calculus

Send and receive primitives are restricted to pass single names.

**Monadic pi-actions**

\[ \pi ::= x\langle y \rangle \quad \text{send name } y \text{ along channel } x \]
\[ | \quad x(y) \quad \text{receive name } y \text{ along channel } x \]
\[ | \quad \tau \quad \text{unobservable action} \]

Monadic processes defined as before on top of monadic pi-actions.
Obvious idea for a translation from Pi to monadic Pi:

\[
\bar{x}\langle \tilde{y} \rangle \rightarrow \bar{x}\langle y_1 \rangle \ldots \bar{x}\langle y_n \rangle \\
\bar{x}(\tilde{y}) \rightarrow \bar{x}(y_1) \ldots \bar{x}(y_n)
\]
Obvious idea for a translation from Pi to monadic Pi:

\[ \overline{x} \langle \tilde{y} \rangle \rightarrow \overline{x} \langle y_1 \rangle \ldots \overline{x} \langle y_n \rangle \]
\[ x(\tilde{y}) \rightarrow x(y_1) \ldots x(y_n) \]

Does not work
Simulating Pi with Monadic Pi

First attempt

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

  \[
  \overline{x} \langle \overline{y} \rangle \rightarrow \overline{x} \langle y_1 \rangle \ldots \overline{x} \langle y_n \rangle \\
  x(\overline{y}) \rightarrow x(y_1) \ldots x(y_n)
  \]

- Does not work

- Counterexample

  \[
  x(y_1 \ y_2) \cdot P \mid \overline{x} \langle z_1 \ z_2 \rangle \cdot Q \mid \overline{x} \langle z'_1 \ z'_2 \rangle \cdot Q'
  \]
Simulating Pi with Monadic Pi
Correct encoding

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

\[
\begin{align*}
\bar{x}⟨\hat{y}⟩. P & \rightarrow (νw) \bar{x}⟨w⟩\bar{w}⟨y_1⟩\ldots\bar{w}⟨y_n⟩. P^† \\
x(\hat{y}). Q & \rightarrow x(⟨w⟩). w(y_1)\ldots w(y_n). Q^†
\end{align*}
\]

where \( P^† \) and \( Q^† \) 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 \( Q_A \) depends on \( A \) and \( P \) is the scope of \( A \). The translation is given by

1. invent 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 \( \bar{a}\langle\tilde{w}\rangle \);
3. replace \( P \) and the old definition of \( A \) by

\[ \hat{P} = (\nu a) (\hat{P} \mid! x(\tilde{x}.\hat{Q}_A)) \]