Concurrency WS 2014/15
Message Passing Concurrency

Peter Thiemann

February 3, 2015

Outline

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

Choice $(a \to P \mid b \to Q)$
await synchronizaton on a or b and continue with P or Q, respectively

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 the other 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 <u>typed bidirectional channels</u>
- Golang’s channels are synchronous

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)
}
```

```
// 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
- `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

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

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

- value-passing CCS

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

- Pi: synchronization on variable events (names) + value 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 (νn) as “new n” (called <u>restriction</u>)

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

- 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

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

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

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

$$B(in, out) = in(x).B(x, in, out)$$

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

- 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

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

Formal Syntax of Pi-Calculus

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

Pi-actions

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

Pi-processes

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

Summation

- In $\sum_{i \in I} \pi_i.P_i$, the process P_i is guarded by the action π_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 $P\{\tilde{z} := \tilde{u}\}$

Summation

- In $\sum_{i \in I} \pi_i.P_i$, the process P_i is guarded by the action π_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

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

Restriction

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

Restriction

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

Example

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

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 <u>replication</u> $!P$ 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 <u>repeatedly</u> send it over y.

Variation: Monadic Pi-Calculus

Send and receive primitives are restricted to pass single names.

Monadic pi-actions

$$\begin{array}{rcll} \pi & ::= & \overline{x}\langle y\rangle & \text{send name } y \text{ along channel } x \\ & \mid & x(y) & \text{receive name } y \text{ along channel } x \\ & \mid & \tau & \text{unobservable action} \end{array}$$

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:

$$\begin{array}{lcl} \overline{x}\langle\tilde{y}\rangle & \to & \overline{x}\langle y_1\rangle \ldots \overline{x}\langle y_n\rangle \\ x(\tilde{y}) & \to & x(y_1)\ldots x(y_n) \end{array}$$

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

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

- Does not work

Simulating Pi with Monadic Pi

First attempt

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

$$\begin{array}{lcl} \overline{x}\langle \tilde{y} \rangle & \rightarrow & \overline{x}\langle y_1 \rangle \ldots \overline{x}\langle y_n \rangle \\ x(\tilde{y}) & \rightarrow & x(y_1) \ldots x(y_n) \end{array}$$

- 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 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 $\overline{a}\langle\tilde{w}\rangle$;
3. replace P and the old definition of A by

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