

Programming Paradigms

Lecture 9-10

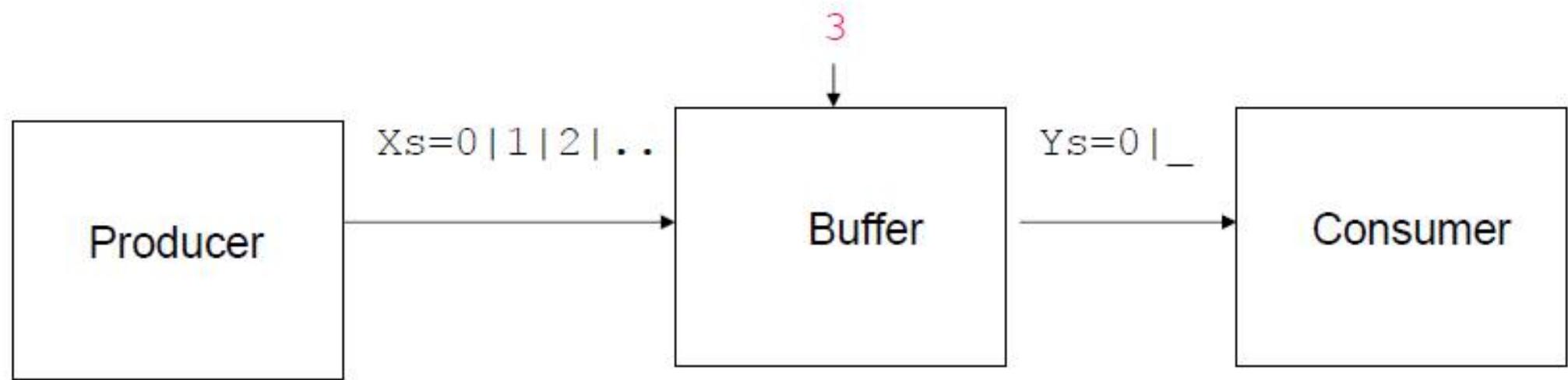
Slides are from Prof. Chin Wei-Ngan from NUS

More on Declarative Concurrency

Bounded Buffer

- *Eager – producer may run ahead*
- *Demand-driven – consumer in control but more complex execution.*
- *Compromise : Bounded Buffer*

Bounded Buffer



$Xs=\{\text{Produce } 0 \text{ Limit}\}$

{Buffer 4 Xs Ys}

S={Consume Ys 0}

```
local Xs Ys S in
  thread {DGenerate 0 Xs} end
  thread {Buffer 4 Xs Ys} end
  thread S={DSum Ys 0 10} end
  {Browse Xs} {Browse Ys} {Browse S}
end
```

```
proc {DGenerate N Xs}
  case Xs of X|Xr then
    X=N
    {DGenerate N+1 Xr}
  end
end
```

```
fun {DSum ?Xs A Limit}
  if Limit > 0 then
    X|Xr=Xs
    in
      {DSum Xr A+X Limit-1}
  else A end
end
```

```
proc {Buffer N ?Xs Ys}
  fun{Startup N ?Xs}
    if N==0 then Xs
    else Xr in Xs=_|Xr {Startup N-1 Xr} end
  end
```

```
proc {AskLoop Ys ?Xs ?End}
  case Ys of Y|Yr then Xr End2 in
    Xs=Y|Xr
    End=_|End2
    {AskLoop Yr Xr End2}
  end
end
```

```
End={Startup N Xs}
in
{AskLoop Ys Xs End}
end
```

Lazy Streams

- Better solution for demand-driven concurrency

Use Lazy Streams

That is consumer decides, so producer runs on request.

Needed Variables

- Idea:
 - start execution,
 - when value for variable needed
 - suspend on the variable
- Value for variable needed...
...a thread suspends on variable!

Lazy Execution (Reminder)

- Up to now the execution order of each thread follows textual order.
Each statement is executed in order strict order, whether or not its results are needed later.
- This execution scheme is called *eager execution*, or *supply-driven* execution
- Another execution order is to execute each statement only if its results are **needed** somewhere in the program
- This scheme is called **lazy evaluation**, or **demand-driven evaluation**

Lazy Execution. Reminder

declare

fun lazy {F1 X} 2*X end

fun {F2 Y} Y*Y end

B = {F1 3}

{Browse B} → nothing (simply unbound B)

C = {F2 4}

{Browse C} → display 16

A = B+C

→ display 6 for B

- F1 is a lazy function
- B = {F1 3} is executed only if its result is needed in A = B+C

Example

```
declare
fun lazy {F1 X} 2*X end
fun lazy {F2 Y} Y*Y end
B = {F1 3}
{Browse B} % → nothing (simply unbound B)
C = {F2 4}
{Browse C} % → nothing (simply unbound C)
```

- F1 and F2 are now lazy functions
- B = {F1 3} and C = {F2 4} are executed only if their results are needed in an expression, like: A = B+C

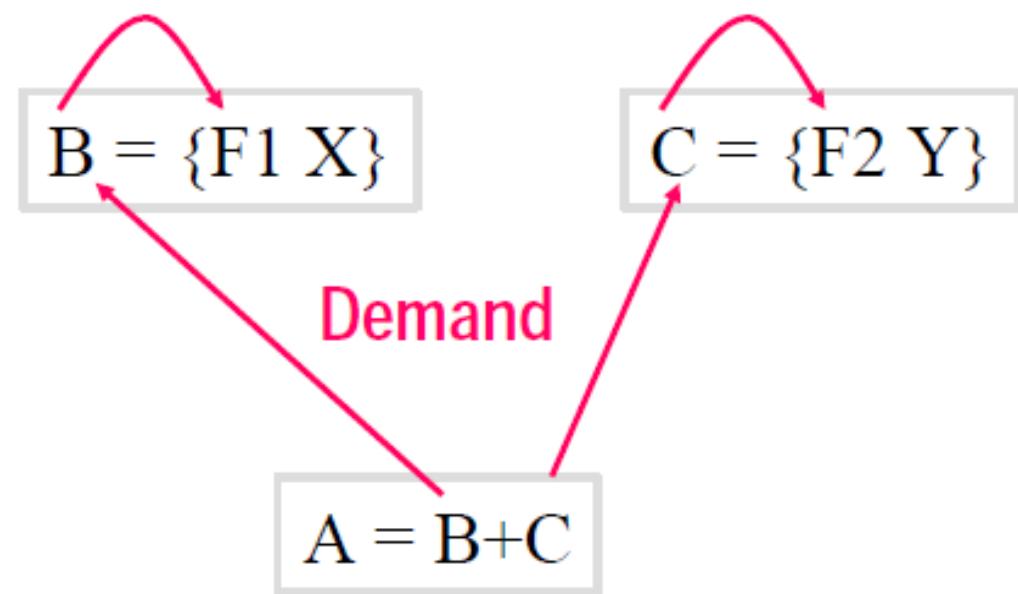
Example

```
declare
fun lazy {F1 X} 2*X end
fun lazy {F2 Y} Y*Y end
B = {F1 3}
{Browse B} % → display 6
C = {F2 4}
{Browse C} % → display 16
A = B+C
```

- F1 and F2 are now lazy functions
- B = {F1 3} and C = {F2 4} are executed because their results are needed in A = B+C

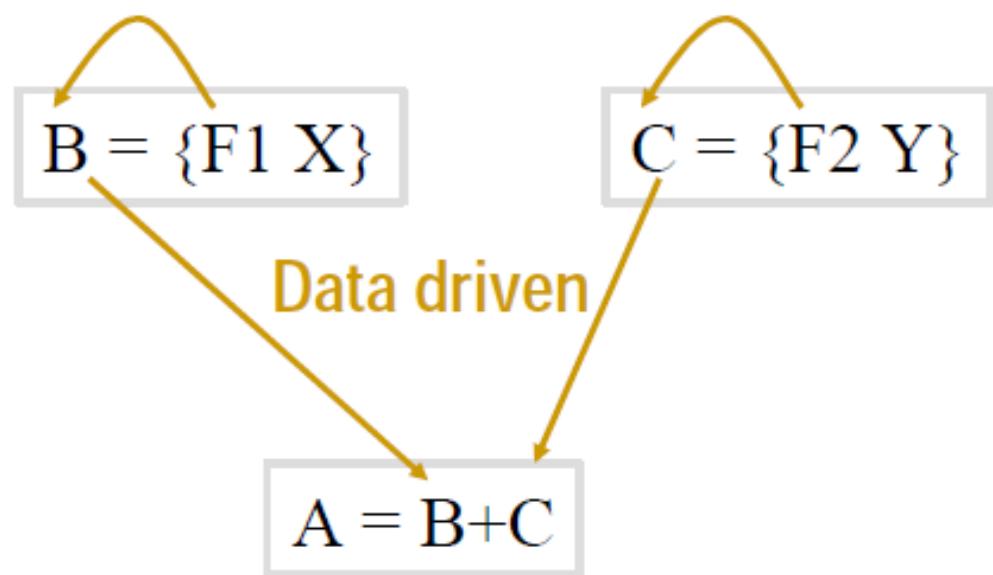
Example

- In **lazy execution**, an operation suspends until its result is needed
- Each suspended operation is triggered when another operation needs the value for its arguments
- In general, multiple suspended operations can start concurrently



Example II

- In **data-driven execution**, an operation suspends until the values of its arguments results are available
- In general, the suspended computation can start concurrently



Triggers

- A by-need trigger is a pair (F, X) :
 - a zero-argument function F
 - a variable X
- Trigger creation
 - $X = \{\text{ByNeed } F\}$ or equivalently
 $\{\text{ByNeed } (\text{proc } \$ A) A = \{F\} \text{ end}) X\}$
- If X is needed, then $X = \{\text{ByNeed } F\}$ means:
 - execute **thread** $X = \{F\}$ **end**
 - delete trigger, X becomes a normal variable

Example 1: ByNeed

```
X={ByNeed fun {$} 4 end}
```

- Executing {Browse X}
 - Shows: X (meaning not yet triggered)
 - Browse does not need the value of X

- Executing T : Z=X+1
 - X is needed
 - current thread T blocks (X is not yet bound)
 - new thread created that binds X to 4
 - thread T resumes and binds Z to 5

Example 2: ByNeed

```
declare
fun {F1 X} {ByNeed fun {$} 2*X end} end
fun {F2 Y} {ByNeed fun {$} Y*Y end} end
B = {F1 3}
{Browse B} % simply display B
C = {F2 4}
{Browse C} % simply display C
```

Example 2: ByNeed

```
declare
fun {F1 X} {ByNeed fun {$} 2*X end} end
fun {F2 Y} {ByNeed fun {$} Y*Y end} end
B = {F1 3}
{Browse B} % display 6
C = {F2 4 }
{Browse C} % display 16
A = B+C
```

Example 3: ByNeed

```
thread X={ByNeed fun {$} 3 end} end  
thread Y={ByNeed fun {$} 4 end} end  
thread Z=X+Y end
```

- Considering that each thread executes atomically, there are six possible executions.
- For lazy execution to be declarative, all of these executions must lead to equivalent stores.
- The addition will wait until the other two triggers are created, and these triggers will then be activated.

Lazy Functions

```
fun lazy {Produce N}  
    N | {Produce N+1}  
end
```

can be implemented with by-need triggers

```
fun {Produce N}  
    {ByNeed fun {$} N | {Produce N+1} end}  
end
```

Lazy Production

```
fun lazy {Produce N}  
    N | {Produce N+1}  
end
```

- Intuitive understanding: function executes only, if its output is needed

Example: Lazy Production

```
fun lazy {Produce N}  
    N | {Produce N+1}  
end  
declare Ns={Produce 0}  
{Browse Ns}
```

- Shows again `Ns`
 - Remember: `Browse` does not need the values of the variables

Example: Lazy Production

```
fun lazy {Produce N}  
    N | {Produce N+1}  
end  
declare Ns={Produce 0}
```

- Execute `_ =Ns.1`
 - needs the variable `Ns`
 - Browser now shows `0 | _` or `0 | <Future>`

Example: Lazy Production

```
fun lazy {Produce N}  
    N | {Produce N+1}  
end  
declare Ns={Produce 0}
```

- Execute _=Ns.2.2.1
 - needs the variable Ns.2.2
 - Browser now shows 0|1|2|_

Everything can be Lazy!

- Not only producers, but also transducers can be made lazy
- Sketch
 - consumer needs variable
 - transducer is triggered, needs variable
 - producer is triggered

Lazy Transducer. Example

```
fun lazy {Inc Xs}
  case Xs
  of X|Xr then X+1|{Inc Xr}
  end
end

declare Xs={Inc {Inc {Produce N}}}
```

Global Summary

- Declarative concurrency
- Mechanisms of concurrent program
- Streams
- Demand-driven execution
 - execute computation, if variable needed
 - need is suspension by a thread
 - requested computation is run in new thread
- By-Need triggers
- Lazy functions

More on Concurrency

Overview

- Stream Object
- Thread Module and Composition
- Soft Real-Time Programming
- Agents and Message Passing
- Protocols
- Erlang

Stream Object

```
input           accumulator           output
      ↓             ↓                   ↑
proc {StreamObject S1 X1 ?T1}
    case S1 of M|S2 then N X2 T2 in
        {NextState M X1 N X2}
        T1 = N|T2 {StreamObject S2 X2 T2}
    [] nil then T1=nil end
end

declare S0 X0 T0
thread {StreamObject S0 X0 T0} end
```

```
StreamObject :: [A], B, [C] → ()
NextState :: A,B, C,A → ()
```

Thread Operations

Common Operations on Thread

{ Thread.this }	return thread id
{ Thread.state T }	return current state of T
{ Thread.suspend T }	suspend T
{ Thread.resume T }	resume T
{ Thread.prompt T }	preempt T
{ Thread.terminate T }	terminate T
{ Thread.injectException T }	raise E in thread T
{ Thread.setPriority T P }	set priority of T
{ Thread.setThisPriority P }	set priority of thread

Common Property Operations

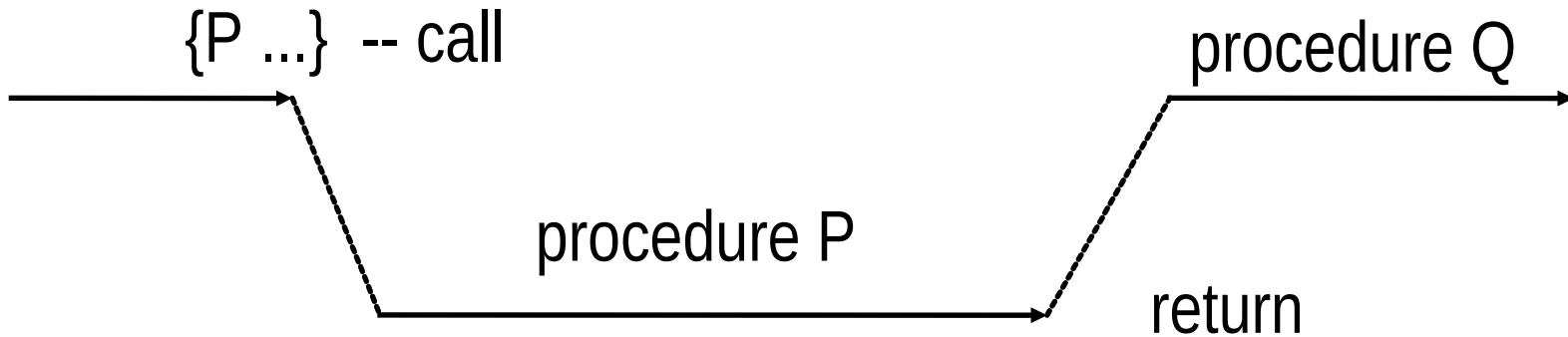
{Property.get priorities} get current priority ratios

{Property.put priorities
p (high:X medium:Y) } set system priority ratios

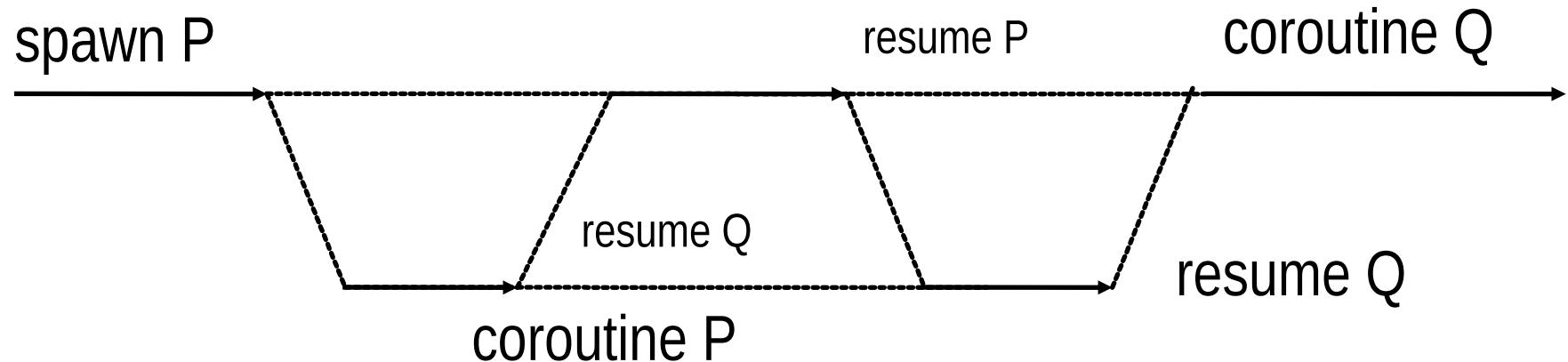
Coroutines

- Languages that do not support concurrent thread might instead support a notion called **coroutining**
- A coroutine is a nonpreemptive thread (sequence of instructions), there is no scheduler
- Switching between threads is the programmer's responsibility

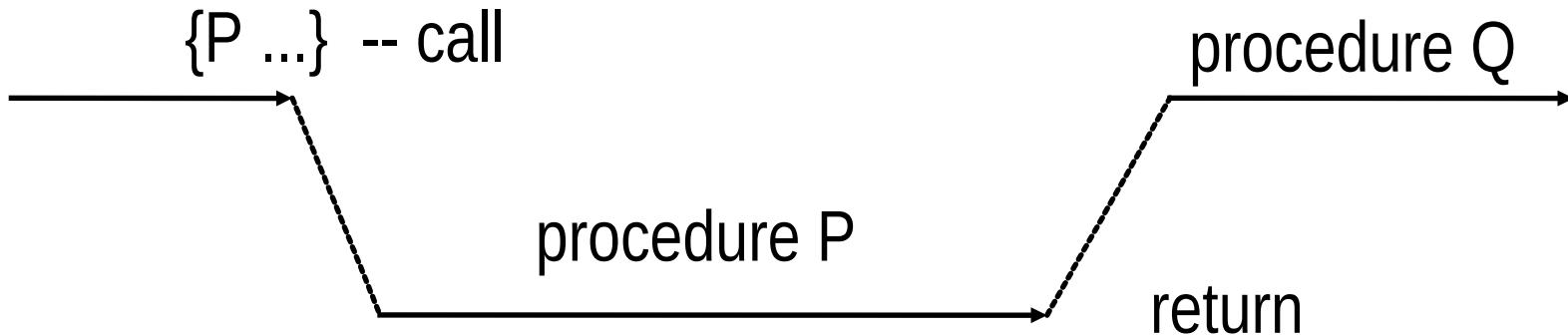
Coroutines



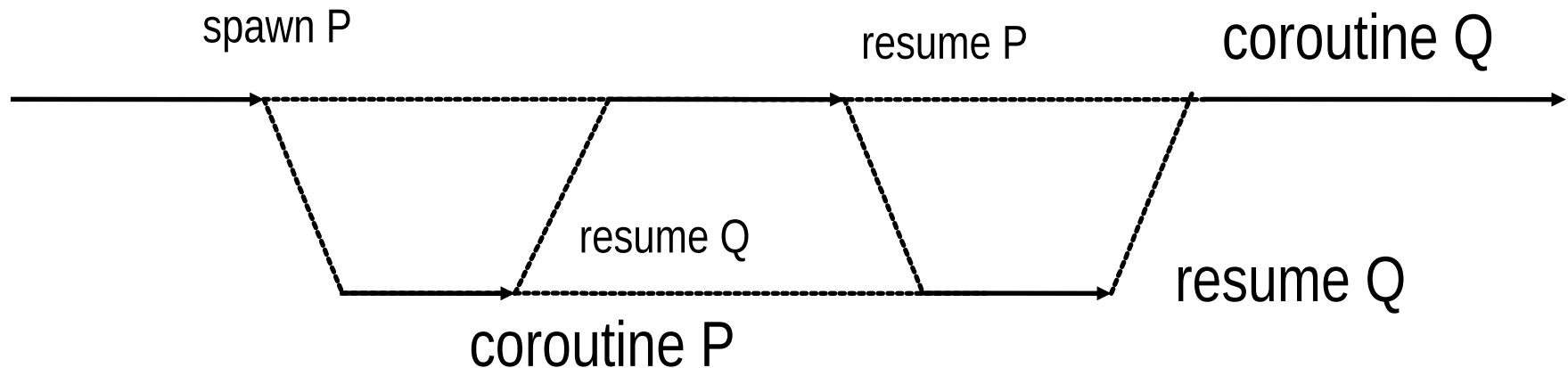
Procedures: one sequence of instructions, program transfers explicitly when terminated it returns to the caller



Coroutines

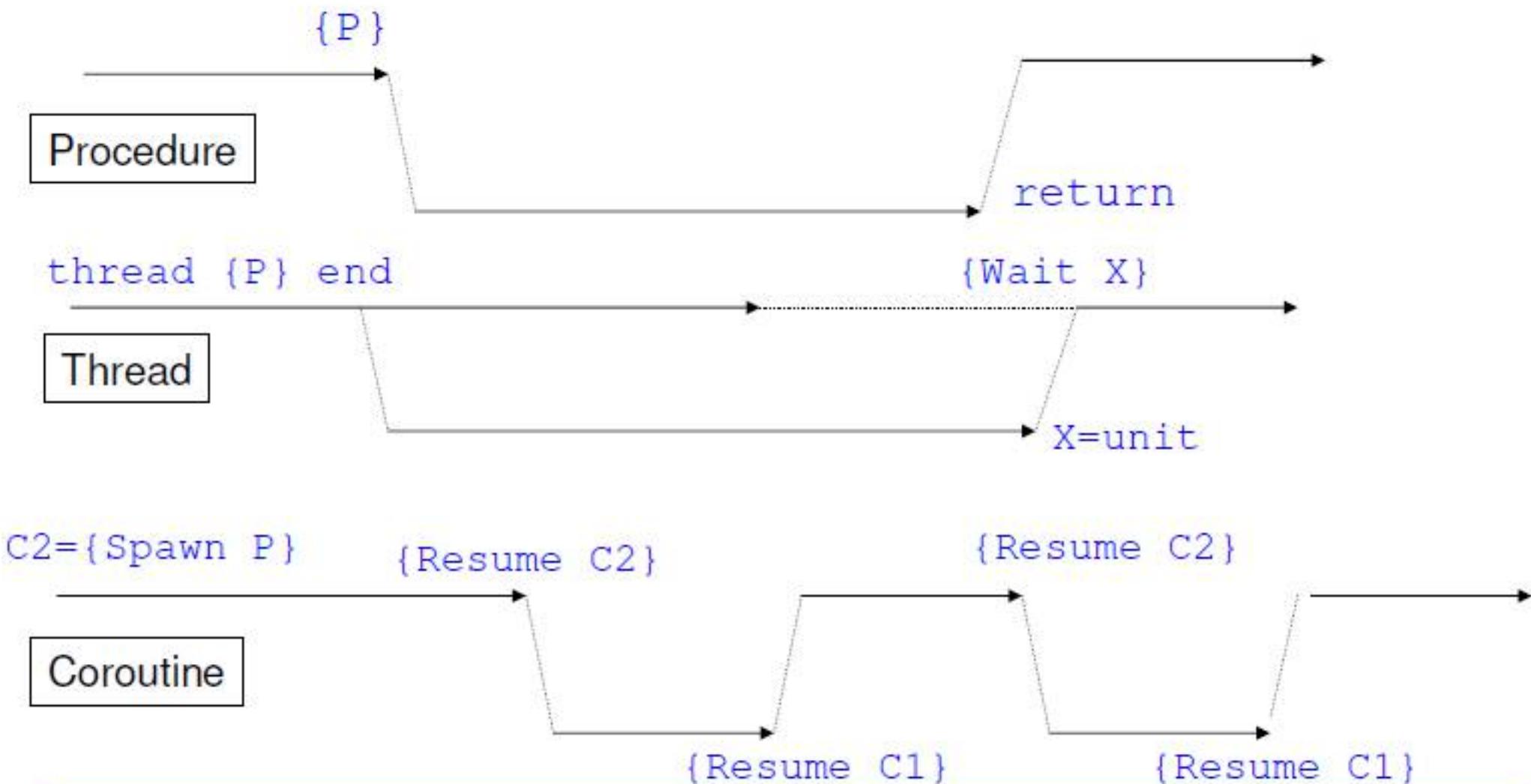


Coroutines: New sequences of instructions, programs explicitly does all the scheduling, by spawn, suspend and resume



Coroutine

A coroutine is a nonpreemptive thread



Basic Mechanism for Coroutines

```
fun {Spawn P}
    PId in
    thread
        Pid={Thread.this}
        {Thread.suspend Pid}
        {P}
```

end

PId

end

Spawn :: () \rightarrow () \rightarrow Id
Resume :: Id \rightarrow ()

```
proc {Resume Id}
    {Thread.resume Id}
    {Thread.suspend {Thread.this} }
end
```

Fork-Join for Threads

```
local X1 X2 .. Xn-1 Xn in
  thread <stmt1> X1=unit end
  thread <stmt2> X2=X1 end
  :
  thread <stmtn> Xn=Xn-1 end
  {Wait Xn}
end
```

wait for all threads to complete through variable binding

Barrier Synchronization

```
list of threads  
proc {Barrier Ps}  
  fun {Loop Ps L}  
    case Ps of P|Pr then M in  
      thread {P} M=L end  
      {Loop Pr M}  
    [] nil then L  
    end  
  end  
  S={Loop Ps unit}  
in  
  {Wait S}  
end  
wait for all threads to complete
```

Soft Real-Time Programming

- Real-time
 - control computations by time
 - animations, simulations, timeouts, ...
- *Hard* real-time has firm deadlines, which have to be respected all the time, without any exception (medical equipments, air traffic control, ...)
- *Soft* real-time is used in less demanding situations.
 - suggested time
 - no time guarantees
 - no hard deadlines as for controllers, etc.
 - Examples: telephony, consumer electronics, ...

The Time module

- The Time module contains a number of useful soft real-time operations:
 - ❑ Delay
 - ❑ Alarm
 - ❑ Time
- {Delay N} suspends the thread for N milliseconds
- Useful for building abstractions
 - ❑ timeouts
 - ❑ repeating actions

The Time module

- `{Alarm N U}` creates a new thread that binds `U` to `unit` after at least `N` milliseconds.
- `Alarm` can be implemented with `Delay`
- `{Time.time}` returns the integer number of seconds that have passed since the current year started

Soft Real-Time Programming. Example

```
functor
import
    Browser(browse:Browse)
define
    proc {Ping N}
        if N == 0 then {Browse 'ping terminated'}
        else {Delay 500} {Browse ping} {Ping N - 1} end
    end
    proc {Pong N}
        {For 1 N 1
            proc {$ I} {Delay 600} {Browse pong} end }
        {Browse 'pong terminated'}
    end
in
    {Browse 'game started'}
    thread {Ping 6} end
    thread {Pong 6} end
end
```

Soft Real-Time Programming. Example



The screenshot shows a window titled "Oz Browser" with a blue header bar. The menu bar contains "Browser", "Selection", and "Options". The main window displays a sequence of messages in a text-based log:

```
'game started'  
ping  
pong  
ping  
pong  
ping  
pong  
ping  
pong  
ping  
pong  
ping  
pong  
'ping terminated'  
pong  
pong  
'pong terminated'  
[ ]
```

The window has standard operating system-style scroll bars on the right side and a small status bar at the bottom.

Agents and Message Passing Concurrency

Client-Server Architectures

- Server provides some service
 - receives message
 - replies to message
 - examples: web server, mail server, ...
- Clients know address of server and use service by sending messages
- Server and clients run independently

Client-Server Applications ...

- With declarative programming, it is impossible to write a client/server program where the server does not know which client will send the next message.
- **Observable *nondeterministic* behavior:** the server can receive information in any order from two independent clients.
- The server has **only** an input stream from which it reads commands.

The Message-Passing Concurrent Model

- Extends the declarative concurrent model by adding one new concept, an **asynchronous communication channel**.
- Any client can send messages to the channel at any time and the server can read all the messages from the channel (no limitations).
- A client/server program may give different results on different executions because the order of clients' sends is not fixed.
- **Message-passing model** is *nondeterministic* and therefore no longer declarative.

Peer-to-Peer Architectures

- Similar to Client-Server:
 - every client is also a server
 - communicate by sending messages to each other
- We call all these guys (client, server, peer)
agent
- In [van Roy, Haridi; 2004] book, this is called
portObject

Common Features

■ Agents

- | | |
|---------------------|-----------------------------|
| ❑ have identity | mail address |
| ❑ receive messages | mailbox |
| ❑ process messages | ordered mailbox |
| ❑ reply to messages | pre-addressed return letter |

■ Now how to cast into programming language?

Message Sending

- Message data structure
- Address **port**
- Mailbox stream of messages
- Reply dataflow variable in message

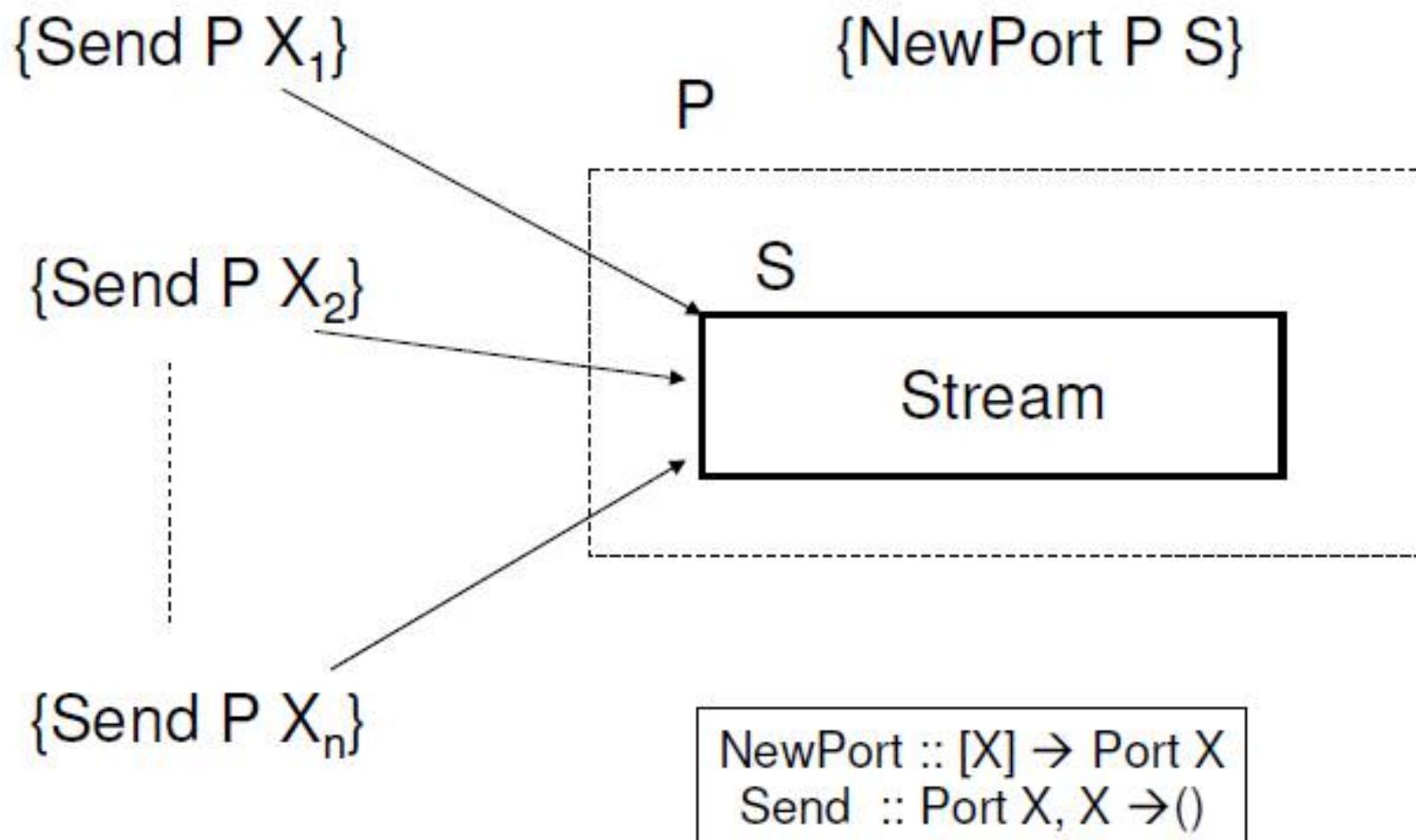
Type :: Port X

message type

Ports

- A **port** is an ADT with two operations:
 - $\{\text{NewPort } S \ P\}$ or equivalently $P = \{\text{NewPort } S\}$: create a new port with entry point (channel) P and stream S .
 - $\{\text{Send } P \ x\}$: append x to the stream corresponding to the entry point P .
- Successive sends from the same thread appear on the stream in the same order in which they were executed.
- This property implies that a port is an asynchronous FIFO (first-in, first-out) communication channel.

Port and its Stream



Ports

- **Asynchronous:** a thread can send a message at any time and it does not need to wait for any reply.
- As soon as the message is in the communication channel, the thread can continue executing.
- Communication channel can contain many pending messages, which are waiting to be handled.

Example

```
declare S P  
P={NewPort S}  
{Browse S}
```

- Displays initially `S<future>` (or `_`)

Example

```
declare S P
```

```
P={NewPort S}
```

```
{Browse S}
```

- Execute {Send P a}
- Shows a |_<future>

Example

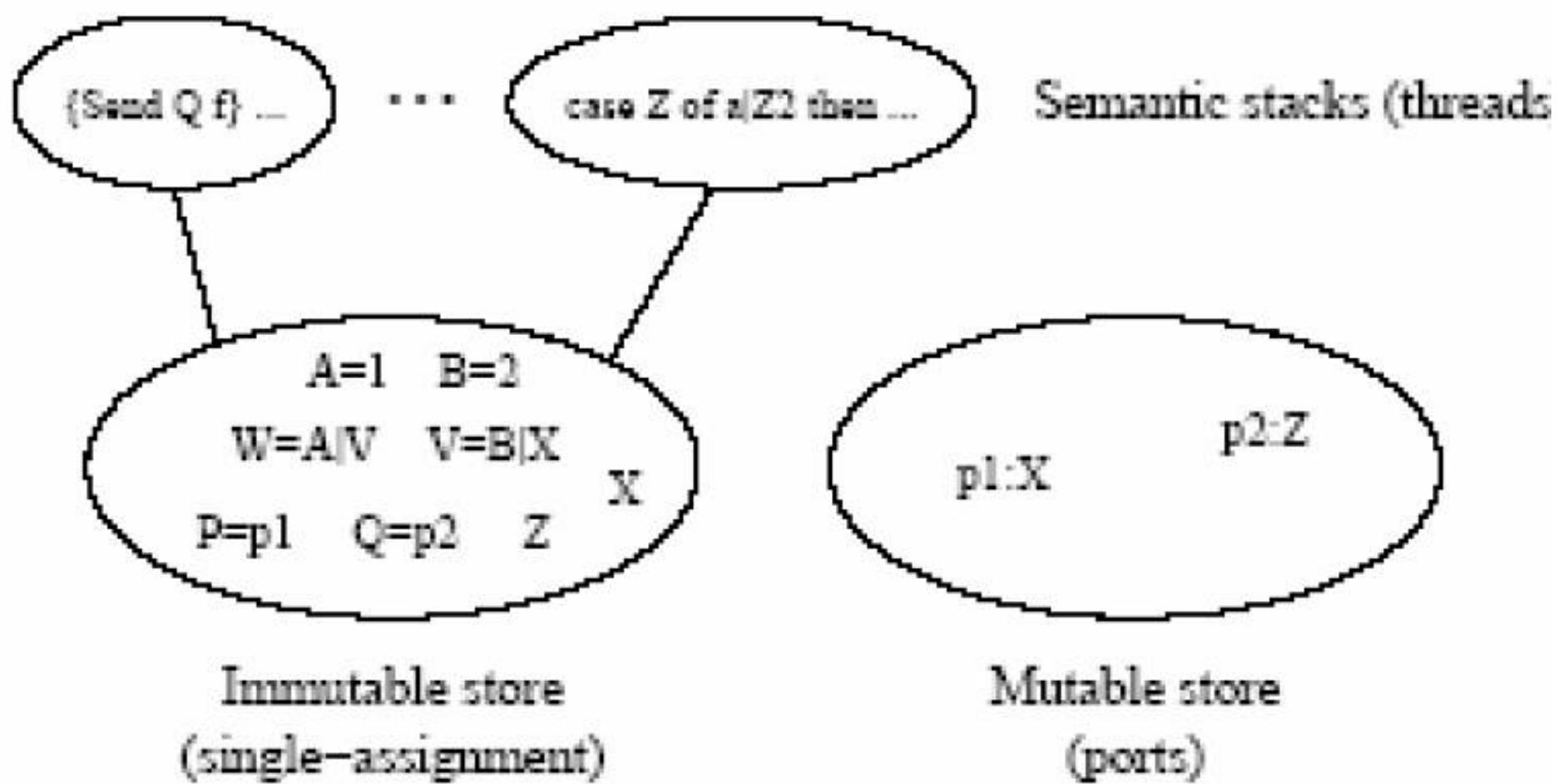
```
declare S P  
P={NewPort S}  
{Browse S}
```

- Execute {Send P b}
- Shows a|b|_<future>
- Note that {Send P a} and {Send P b} are in the same thread

Semantics of Ports

- Extend the execution state of the declarative model by adding a **mutable store** μ
- This store contains ports, i.e. pairs of the form $x : y$, where x and y are variables of the single-assignment store (x is the channel's name and y is the *current last position of stream*).
- The mutable store is initially empty.
- The semantics guarantees that x is *always bound* to a name value that represents a port and that y is *unbound*.
- The execution state becomes a triple (MST, σ, μ) (or (MST, σ, μ, τ) if the trigger store is considered).

The Message-Passing Concurrent Mode



The NewPort Operation

- The semantics of $(\{\text{NewPort } \langle x \rangle \langle y \rangle\}, E)$ is:
 - Create a **fresh port name** (also called **unique address**) n .
 - Bind $E(\langle y \rangle)$ and n in the store.
 - If the binding is successful, then add the pair $E(\langle y \rangle) : E(\langle x \rangle)$ to the mutable store μ .
 - If the binding fails, then raise an error condition.

The Send Operation

- The semantics of $(\{\text{Send } \langle x \rangle \langle y \rangle\}, E)$ is:
 - If the activation condition is true ($E(\langle x \rangle)$ is determined), then:
 - If $E(\langle x \rangle)$ is not bound to the name of a port, then raise an error condition.
 - If the mutable store contains $E(\langle x \rangle) : z$, then:
 - Create a new variable $z0$ in the store.
 - Update the mutable store to be $E(\langle x \rangle) : z0$.
 - Create a new list pair $E(\langle y \rangle) | z0$ and bind z with it in the store.
 - If the activation condition is false, then suspend execution.

Question

```
declare S P  
P={NewPort S}  
{Browse S}  
thread {Send P a} end  
thread {Send P b} end
```

- What will the Browser show?
- Note that each `{Send P ...}` is in a separate thread

Question

```
declare S P  
P={NewPort S}  
{Browse S}  
thread {Send P a} end  
thread {Send P b} end
```

- Which will the Browser show?
- Either
 - $a \mid b \mid _ <future>$ or
 - $b \mid a \mid _ <future>$
- non-determinism: we can't say what

Answering Messages

- Traditional view
- Include the entry port P' of the sender in the message:
 $\{\text{Send } P \text{ pair(Message } P')\}$
- Receiver sends answer message to P'
 $\{\text{Send } P' \text{ AnsMessage}\}$

Answering Messages

- Do not reply by address, use something like pre-addressed reply envelope
 - dataflow variable!!!
- {Send P pair(Message Answer) }
- Receiver can bind Answer!