Programación Concurrente y Distribuida
Paso de mensajes

Camilo Rueda [1]

[1] Universidad Javeriana-Cali

PUJ 2010

Hacia concurrencia de paso de mensajes

Ejercicio: Modelar captura de un bus

- Un cierto número de procesos
- asíncronamente tratan de capturar un Bus
- El Bus está controlado por un proceso
- El primer proceso que solicite el bus, lo obtiene

Concurrencia de paso de mensajes

En concurencia declarativa:

- Comunicación es síncrona
- Recuerde objetos stream:

```
proc {StreamObject S1 X1 ?T1 }
  case S1
  of M | S2 then N X2 T2 in
    {ProxEstado M X1 N X2}
    T1=N |T2
    {StreamObject S2 X2 T2}
  [] nil then T1=nil end
end
```

- Se puede encapsular como estructura de datos?

Programar sistema de cliente/servidor declarativo

- Opciones propuestas:
  - Cada cliente envía mensajes independientes al servidor (ver pasoMensajes1.oz clienteServidor_modelo1)
  - Imponer sincronización fuerte entre los clientes (ver pasoMensajes1.oz clienteServidor_modelo2)

Conceptos

- Una idea nueva: canal de comunicación asíncrono
  i.e Los agentes no esperan por una respuesta
- El canal es un puerto:
  - Con un stream asociado
    objeto puerto=puerto+stream
  - El objeto puerto lee mensajes de su stream y envía mensajes a otros puertos
  - Cada objeto puerto se define mediante un procedimiento recursivo declarativo

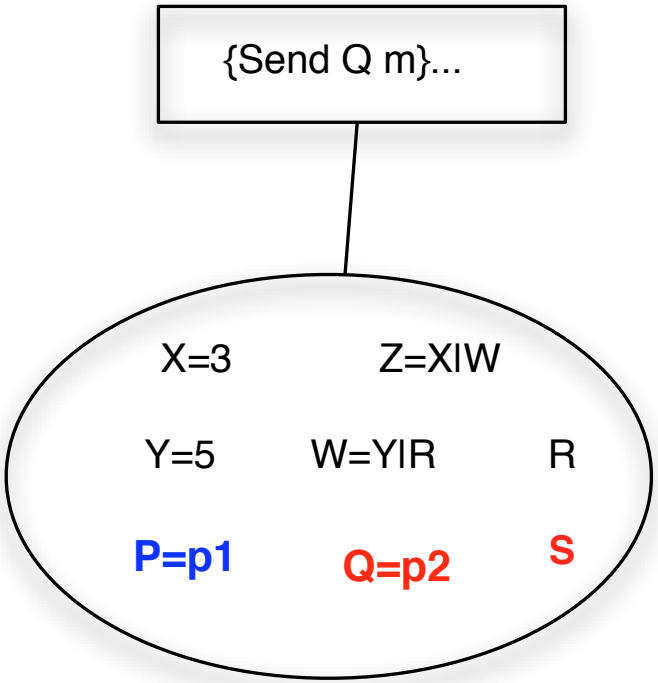
Puerto

- Las variables de asignación única pueden referenciar puertos
- Un puerto es un nombre único que se guarda en un store mutable
- al puerto se asocia un stream, referenciado por el nombre único (su dirección) $n : S$
  - la dirección n es única
  - el store de asignación única contiene los cambios (mensajes) en el stream a través del tiempo
- El puerto referencia siempre la cola del stream de mensajes
- Enviar un mensaje M al puerto redefine la cola del stream como una lista cuyo primer elemento es M

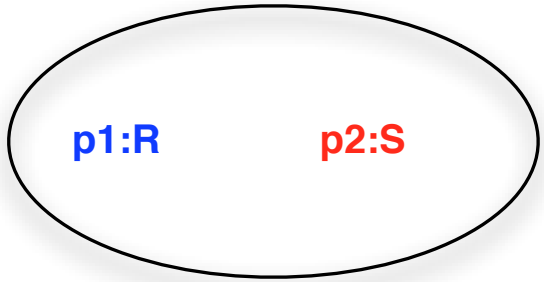
Enviar mensaje

- Un puerto $p2 : S$
- Enviar M a $p2$:
  - Lee el stream guardado en la dirección $p2$
  - crea una nueva variable S' en el store
  - Impone $S = M \mid S'$
  - En el store mutable, asigna a la dirección $p2$ el stream S' $p2 : S'$

Store mutable

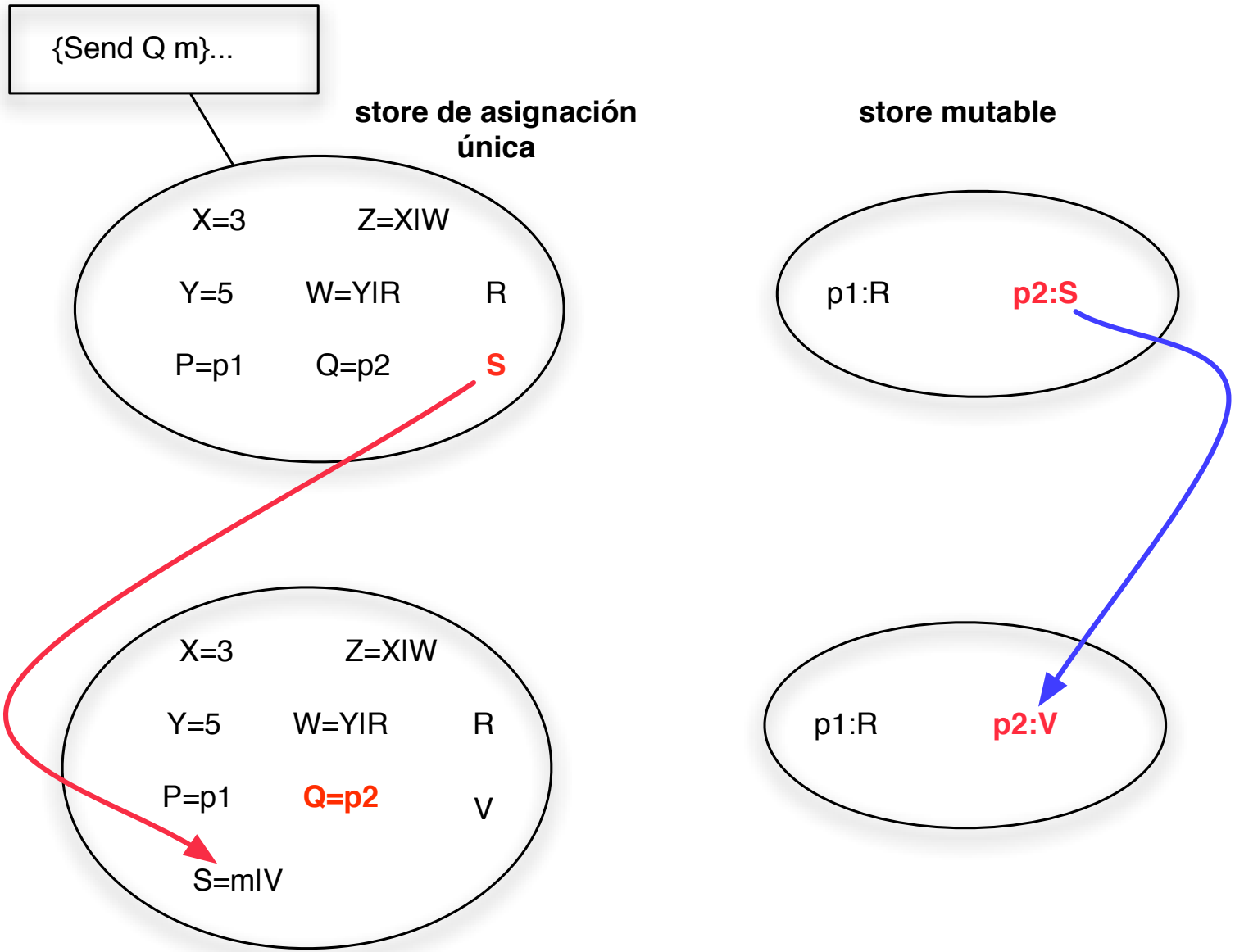


store de asignación única



store mutable

Store mutable



El lenguaje de núcleo

$< s >::= skip$	*stmt vacío*
$\mid < s1 >< s2 >$	*secuencia*
$\mid local\ x\ in\ s\ end$	*creación de var.*
$\mid x1 = x2$	*Var-var binding*
$\mid x = v$	*creación de valor*
$\mid if\ x\ then\ s1\ else\ s2\ end$	*Condicional*
$\mid case\ x\ of$ $< pattern >\ then\ s1\ else\ s2\ end$	*Pattern matching*
$\mid \{x\ y_1 ... y_n\}$	*aplicación de procedimiento*
$\mid thread\ < s >\ end$	*creación de un hilo*
$\mid \{ByNeed\ < x >\ < y >\}$	*creación de un disparador*
$\mid \{NewPort\ < y >\ < x >\}$	*crear puerto*
$\mid \{Send\ < x >\ < y >\}$	*enviar a puerto*

El tipo abstracto puerto

- Creación de un puerto:
  {**NewPort** *Xs P*}
  - P es una variable de tipo puerto
  - *P* queda asignado a un nombre nuevo *d*
  - *d* queda asociado con *Xs* en el store mutable
- Enviar mensaje
  {**Send** *P M*}

Semántica formal de *NewPort*

La instrucción semántica $(\{NewPort < x > < y >\}, E)$:

- Crea un nombre nuevo n (la dirección del puerto)
- Liga $E(< y >)$ con n en el store
- Si lo anterior tiene éxito, agrega el par $E(< y >) : E(< x >)$ al store mutable
- Si la ligadura falla, levanta una excepción de error.

Semántica de *Send*

La instrucción $(\{Send\ < x >< y >\}, E)$:

- Si la condición de activación es **true** (i.e $E(< x >)$ está determinado), entonces:
  - Si $E(< x >)$ no está ligado al nombre de un puerto, levanta excepción de falla
  - Si el store mutable contiene $E(< x >) : z$ entonces:
    - Crea una variable nueva z' en el store
    - Actualiza el store mutable con $E(< x >) : z'$
    - Crea una nueva lista $E(< y >) \mid z'$ y la liga con z en el store
- Si la condición de activación es **false**, suspende la ejecución

Objetos puerto

Combinación de uno o más puertos, con un stream

- Crea los puertos (globales) y los stream (locales)
- Ejecuta un procedimiento recursivo sobre los streams

```
declare  P1 P2 ... Pn
local    S1 S2 ... Sn in
         {Newport S1 P1}
         {Newport S2 P2}
         ...
         {Newport Sn Pn}
         thread {Rec S1 S2 ... Sn} end
end
```

Objeto puerto con estado

```
fun   {NuevoObjetoPuerto Init Fun}
      proc {CicloMsg S1 Estado}
       case S1 of Msg|S2 then
           {CicloMsg S2 {Fun Msg Estado}}
        [] nil then skip end
      end
      Sin
in
      thread {CicloMsg Sin Init} end
      {Newport Sin}
end
```

Objeto puerto sin estado

```
fun   {NuevoObjetoPuerto2 Proc}
      Sin in
       thread for Msg in Sin do {Proc Msg} end  end
       {NewPort Sin}
end
```

(ver puertos_clienteServidor en pasoMensajes1.oz)

Protocolos

- Objetos Puerto operan intercambiando mensajes de forma coordinada
- Protocolo: una secuencia de mensajes entre una o más partes que puede entenderse “por encima” de la noción de mensaje.
- Protocolo simple: RMI
  - “método”: Comportamiento del puerto cuando recibe un mensaje
  - El cliente envía una petición al servidor y espera a que el servidor envíe un reconocimiento.

RMI síncrono: servidor

```
proc {ProcServidor Msg}
   case Msg of calc(X Y)
   then
      Y = X * X + 3.0 * (X + 1.0)
   end
end
Server = {NuevoObjetoPuerto2 ProcServidor}
```

RMI síncrono: cliente

```
proc {ProcCliente Msg}
 case Msg of arranque(Y) then Y1 Y2 in
    {Send Server calc(10,0 Y1)} {Wait Y1}
    {Send Server calc(20,0 Y2)} {Wait Y2}
     Y = Y1 + Y2
     end
end
Client = {NuevoObjetoPuerto2 ProcCliente}
{Browse {Send Client arranque($)}}
```