Aula 7

Expressividade do CCS e CCS com passagem de valores

Algoritmo de Peterson para a Exclusão Mútua

- P_1, P_2 processos
- variáveis partilhadas b_1, b_2 e k, sendo
- se $k = i$ então P_i pode entrar
- P_1 faz$k = 2$ (dá o privilégio a P_2)
- e simétricamente para P_2
- $b_i = true$ quando P_i espera
- $b_i = false$ quando P_i sai da zona crítica.

Processo P_i
while true do
noncricital actions
$b_i \leftarrow$ **true;**
$k \leftarrow j$**;**
while $b_j \wedge k = j$ **do**
skip;
critical actions
$b_i \leftarrow$ **false**;

Algoritmo de Peterson em CCS

- As variáveis são processos cujos estados são os seus possíveis valores
- Para b_1 temos estados B_{1t} e B_{1f}
- Outros processos podem ler ou escrever o valor de variáveis comunicando com o respectivo processo o valor pretendido
- Para um processo ler **true** em b_1 sincroniza com esse processo por uma ação (canal) p.e $b1rt$
- Para um processo escrever **false** em b_1 sincroniza com esse processo por uma ação (canal) p.e $b1wf$

Processos para as variáveis

$$\begin{aligned} B_{1f} &:= b1rf!.B_{1f} + b1wf?.B_{1f} + b1wt?.B_{1t} \\ B_{1t} &:= b1rt!.B_{1t} + b1wf?.B_{1f} + b1wt?.B_{1t} \end{aligned}$$

$$\begin{aligned} B_{2f} &:= b2rf!.B_{2f} + b2wf?.B_{2f} + b2wt?.B_{2t} \\ B_{2t} &:= b2rt!.B_{2t} + b2wf?.B_{2f} + b2wt?.B_{2t} \end{aligned}$$

$$\begin{aligned} K_1 &:= kr1!.K_1 + kw1?.K_1 + kw2?.K_2 \\ K_2 &:= kr2!.K_2 + kw1?.K_1 + kw2?.K_2 \end{aligned}$$

Processos para P_1 e P_2

- apenas modelar a entrada e saída da zona crítica
- supomos que não podem terminar na zona crítica ou ficar lá para sempre
- para representar o ciclo **while** para P_1 temos um estado P_{11} tal que
  - ler os valores de b_2 e k
  - esperar se $b_2 = \mathbf{true} \wedge k = 2$
  - mudar de estado, P_{12}
  - em P_{12} entrar e sair da zona critica
  - mas como avaliar $b_j \wedge k = j$?
  - supomos da esquerda para a direita e a segunda não é avaliada se a primeira for falsa

Processos para P_1 e P_2 em CCS

$$\begin{aligned} P_1 &:= b1wt!.kw2!.P_{11} \\ P_{11} &:= b2rf?.P_{12} + b2rt?.(kr2?.P_{11} + kr1?.P_{12} \\ P_{12} &:= enter1.exit1.b1wf!.P_1 \end{aligned}$$

$$\begin{aligned} P_2 &:= b2wt!.kw1!.P_{21} \\ P_{21} &:= b1rf?.P_{22} + b1rt?.(kr1?.P_{21} + kr2?.P_{22} \\ P_{22} &:= enter2.exit2.b3wf!.P_2 \end{aligned}$$

Algoritmo de Peterson em CCS

Se $k = 1$ no inicio

$$Peterson \quad := \quad (P_1|P_2|B_{1f}|B_{2f}|K_1)\backslash L$$

onde L são todas as ações excepto as de entrada e saída.

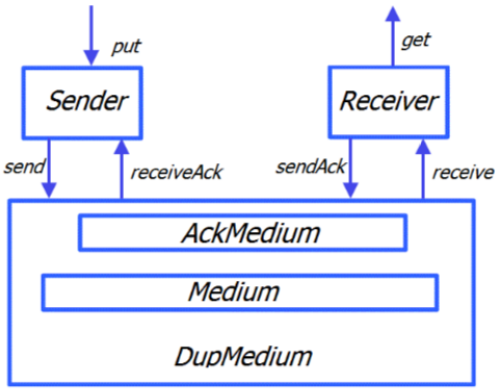
Exclusão mutua em CCS

Se se modelar a entrada e a saída da zona crítica fica

$$MutexSpec \quad := \quad enter1.exit1.MutexSpec + enter2.exit2.MutexSpec$$

Será que $MutexSpec \approx Peterson$?

Exemplo de um Protocolo com erro no meio –Pseuco



$$\begin{aligned}
Sender &:= put?.send!.Sending \\
Sending &:= receiveAck?.Sender + receiveNAck?.send!.Sending \\
Receiver &:= receive?.get?.sendAck!.Receiver + \\
&\quad gargled?.sendNAck!.Receiver \\
Medium &:= send?.(receive!.Medium + i.garbled!.Medium) \\
AckMedium &:= sendAck?.receiveAck!.AckMedium + \\
&\quad sendNAck?.receivedNAck!.AckMedium \\
DupMedium &:= Medium|AckMedium \\
Protocol &:= (Sender \mid Receiver \mid DupMedium)\backslash \\
&\quad \{send, receive, sendAck, receiveAck, \\
&\quad receiveNAck, sendNAck, garbled\}
\end{aligned}$$

CCS com passagem de valor

$$
\begin{array}{rcl}
Sender & := & put?x.send!x.Sending[x] \\
Sending[x] & := & receiveAck?.Sender + receiveNAck?.send!x.Sending[x] \\
Receiver & := & receive?x.get?x.sendAck!.Receiver + \\
 & & gargled?.sendNAck!.Receiver \\
Medium & := & send?x.(receive!x.Medium + i.garbled!.Medium) \\
AckMedium & := & sendAck?.receiveAck!.AckMedium + \\
 & & sendNAck?.receivedNAck!.AckMedium \\
DupMedium & := & Medium|AckMedium \\
Protocol & := & (Sender \mid Receiver \mid DupMedium)\backslash \\
 & & \{send, receive, sendAck, receiveAck, \\
 & & receiveNAck, sendNAck, garbled\}
\end{array}
$$

$Protocol|put!1.put!2.put!3.put!4.0\backslash\{put\}$

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)
- ou usar variáveis
- $a!x$: saída do valor guardado em x no canal a (enviar)
- $a?x$: entrada de um valor que se guarda em x pelo canal a (receber)
- e os nomes podem ter variáveis como parametros permitindo assim enviar e receber valores ($A[x, y]$)

CCS_{vp} com passagem de valor

Sendo $\mathbb{V}$ um conjunto de valores e $\mathbb{K}$ um conjunto de canais temos

$$
\begin{array}{rcl}
A^! & = & \{a!v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a! \mid a \in \mathbb{K}\}, \\
A^? & = & \{a?v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a? \mid a \in \mathbb{K}\}, \\
Com & = & A^! \cup A^? \\
Act & = & Com \cup \{\tau\}
\end{array}
$$

$$
\begin{array}{rcl}
P & ::= & 0 \mid X[r_1,\ldots,r_n] \mid P+P \mid \chi.P \mid P|P \mid P\backslash H \\
\chi & ::= & \tau \mid a! \mid a? \mid a!v \mid a?v \mid a!x \mid a?x
\end{array}
$$

onde $X \in Var$, $x \in D$, $r_i \in D \cup \mathbb{V} \cup \mathbb{K}$

Regras do CCS_{vp}

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$
\begin{array}{rcll}
(a!y.P)\{v/x\} & = & a!y.P\{v/x\} & \text{se } y \neq x \\
(a!x.P)\{v/x\} & = & a!v.P\{v/x\} & \\
(a?y.P)\{v/x\} & = & a?y.(P\{v/x\} & \text{se } y \neq x \\
(a?x.P)\{v/x\} & = & a?x.P & \\
X[x]\{v/x\} & = & X[v] & \\
X[y]\{v/x\} & = & X[y] & \text{se } y \neq x
\end{array}
$$

Para as restantes expressões é passado para as subexpressões. $put?x : 0..9.send!x.0$

$$\text{Rec}\ \frac{P\{v_1/r_1,\ldots,v_n/r_n\} \xrightarrow{\alpha} P' \qquad \Gamma(X[r_1,\ldots,r_n]) = P}{X[r_1,\ldots,r_n] \xrightarrow{\alpha} P'}$$

$$
\begin{array}{rcl}
send!y.Sending[x]\{3/x,5/y\} & = & send!5.Sending[3] \\
send!y.Sending[x]\{3/x,5/y,receice/send\} & = & receive!5.Sending[3]
\end{array}
$$

when: **Condicional bloqueador**

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$$B[x] \quad ::= \quad when(x<4)put?.B[x+1] + when(x>0)get?.B[x-1]$$

$B[0]$ ou $B[5]$ o que fazem?

$$
\begin{array}{rcl}
send!(x+y)3/x,5/y & = & sent!(8) \\
3+5 & \Downarrow & 8
\end{array}
$$

- $when(b)P$ se b é verdade comporta-se como P senão bloqueia.
- avaliação de expressões $e \Downarrow z$: a expressão e avalia para z

$$P \quad ::= \quad 0 \mid X[r_1, \ldots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P\backslash H \mid when(b)P$$

$$\begin{array}{lcl} IterMult[z, x, y] & := & when(x > 0)i.IterMult[z + y, x - 1, y] \\ & & +when(x == 0)println!z.0 \\ IterMult[0, 3, 7] & & \end{array}$$

Regras CCS^Z_{vp}

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

$$\text{Rec}\ \frac{P\{v_1/r_1, \ldots, v_n/r_n\} \xrightarrow{\alpha} P' \qquad \Gamma(X[r_1, \ldots, r_n]) = P}{X[r_1, \ldots, r_n] \xrightarrow{\alpha} P'}$$

$$\text{Output}\ \frac{e \Downarrow z}{a!e.P \xrightarrow{a!.z} P}$$

$$\text{Valor}\ \frac{e \Downarrow z}{\alpha?e.P \xrightarrow{a?z} P}$$

$$\text{cond}\ \frac{P \xrightarrow{\alpha} P' \qquad b \Downarrow True}{when(b)P \xrightarrow{\alpha} P'}$$

Células de Memória

$$\begin{array}{lcl} Cell[rd, wr, x] & := & rd!x.Cell[rd, wr, x] + wr?y.Cell[rd, wr, y] \end{array}$$

- $Cell[rd, wr, 5]$
- $Cell[rdA, wrA, 0]|Cell[redB, wrB, 0]$

$$
\begin{aligned}
Cells &:= Cell[rdA, wrA, 0] | Cell[redB, wrB, 0] \\
Serve &:= mult?.rdA?x : R.rdB?y : R.IterMult[0, x, y] \\
IterMult[z, x, y] &:= when(x > 0) i.IterMult[z + y, x - 1, y] \\
&\quad + when(x == 0) println!z.Serve \\
Use &:= wrA!7.wrB!5.mult!.0
\end{aligned}
$$

$(Cells|Serve|Use)\backslash rdA, wrA, rdB, wrB, mult$ Qual o resultado?

Factorial...como sempre

$$
\begin{aligned}
Fac[n, j] &:= when(j > 0) i.Fac[n * j, j - 1] \\
&\quad + when(j == 0) println!n.0
\end{aligned}
$$

$Fac[1, 5]$

$$
\begin{aligned}
Fak &:= rdJ?j : R.(when(j > 0) rdN?n.wrN!(n * j).wrJ!(j - 1).Fak + when(j == 0) rdN?n.prin \\
Cell[v, rd, wr] &:= rd!v.Cell[v, rd, wr] + wr?x : R.Cell[x, rd, wr] \\
Cells &:= Cell[0, rdN, wrN] | Cell[0, rdJ, wrJ]
\end{aligned}
$$

$(wrN!1.wrJ!5.Fak|Cells)\backslash rdN, wrN, rdJ, wrJ$

O CCS_{vp} pode ser embebido no CCS

..logo é só "syntatic sugar"...

CCS_{vp}	CCS
$a!v.P$	$a_v!.P$
$a?x.P$	$\sum_{v \in \mathbb{V}} a_v?.P\{v/x\}$
$X[u_1, \ldots, u_n]$	$X_{u_1, \ldots, u_n}$

Isto é basta usar ações e nomes indexados (podendo ser considerados conjuntos de indices infinitos ($\mathbb{V}$ ou D)

PseuCo: Linguagem de programação concorrente

- Semântica formal traduz-se para CCS e é aí interpretada
- Tem uma semântica executável que é implementada num compilador para Java
- Processos são agentes havendo sempre o `mainAgent`

What's new?

```
mainAgent { int n, j;
n = 1;
for (j = 5; j > 0 ; j--){
n = n*j;
}
println (" O factorial de 5 e " + n + ".");
}
```

$$
\begin{array}{rcl}
MainAgent[a] & := & MainAgent_2[a, 5, 1] \\
MainAgent_1[a, \$j, \$n] & := & MainAgent_2[a, \$j - 1, \$n] \\
MainAgent_2[a, \$j, \$n] & := & when(!(\$j > 0))MainAgent_3[a, \$j, \$n] \\
 & & +when(\$j > 0)i.MainAgent_1[a, \$j, \$n * \$j] \\
MainAgent_3[a, \$j, \$n] & := & println!(\text{"}O\ factorial\ de\ 5\ é\ \text{"}^{\$}n\text{"}.\text{"}).0
\end{array}
$$

$MainAgent[1] \backslash *, println, exception$

Tipos de comunicação entre Agentes

- Por variáveis partilhadas: memória partilhada
- Síncrona via mensagens: canais de capacidade 0
- Assíncrona via mensagens: canais de capacidade > 0 (buffers)

Sincronização

```
void factorial(int z, intchan c) {
    int j, n=1;
    for (j = z; j > 0 ; j--) {
        n= n*j;  }
    c <! n; }
mainAgent {
    intchan cc;
    int mid, fin;
    agent a1 = start(factorial(3, cc));
    println("Agent 1 is working for me.");
    mid = <? cc;
    agent a2 = start(factorial(mid, cc));
    println("Agent 2 is working for me.");
    fin = <? cc;
    println("The factorial of the factorial of
      three is " + fin + ".");}
```

Crash course PseuCO

- variáveis podem ser declaradas: locais ou globais
- instruções terminam com ;
- procedimentos tem um valor de retorno ou são de tipo `void`
- condicionais: **if**; operadores $!, ==, \&\&, ||$
- condicionais inline : $n > 5?"mais";"menos"$
- ciclos: **for** e **while**
- estruturas: **struct**
- Em procedimentos
- call-by-value: tipos simples
- call-by-reference: arrays, struct, monitor, mutex e canais

Agentes

- `agent a1=start(<instrucao>)`
- `start(<instrucao);`
- Esperar pela terminação: `join(a1)`

Exemplo

```
int n;
void counter(){
int loop;
for (loop = 0; loop < 5; loop++){
n = n - 1; }
}
mainAgent { n = 10;
agent a1 = start(counter());
agent a2 = start(counter());
join(a1);
join(a2);
println ("The value is "+ n);
}
```

Canais

- `boolchan chan1;` : sincrono
- `intchan7 chan2` : asíncrono
- canais são FIFO: first-in-first-out
- `chan2 <! 7` : envia 7
- `<?chan2` : recebe 7
- `int x=<? chan2`: x fica com 7
- se vazios não enviam
- se se tentar receber de um canal vazio, quem *recebe fica bloqueado*
- se se tentar enviar para um canal cheio, quem *envia fica bloqueado*

`Select-case`

- Vários `case` e um `default`
- não deterministicamente escolhe um que não esteja bloqueado
- `default` nunca está bloquado

```
select {
case chan1 <! a: {
// varias instrucoes}
case b = <? chan2:
// so uma
case <? chan3:
// recebe e esquece
default:
//sempre disponivel
}
```

Memória partilhada

- agentes podem partilhar variáveis
- se simples têm de ser globais
- as estruturas podem ser partilhadas argumentos devido ao call-by-refernce
- *Race-condition/Data-race*: quando um agente pode escrever numa variável que outro está a let
- *Data-race*: programas são incorrectos pelo que sem de evitar isso
- Usar **lock** e **monitor** para garantir a exclusão mútua.

lock

- Permitem coordenar o acesso a variáveis evitando o acesso concorrente e assim o *data race*.
- `lock` : bloqueia
- `unlock`: liberta

```
lock m;
int i=5;
void dec() {
lock(m);
i--;
unlock(m);
}
```

monitor

- É um struct especial
- tem lock implícito: **lock** é usado antes de qualquer acesso a um método do monitor e **unlock** quando o método retorna.

```
monitor count{
int a=0;
int plusPlus)(){
a++;
return a;
}}
```

Contador

```
int n;
lock guard n;
void counter(){
int loop;
for (loop = 0; loop < 5; loop++){
lock(guard n);
n = n -1; unlock(guard n);
} }
mainAgent { n = 10;
agent a1 = start(counter());
agent a2 = start(counter());
```

```
join(a1);
join(a2);
println ("The value is " + n);
}
```

monitor

- Os monitores podem esperar por certas condições (**condition**)
- `waitForCondition` : só procede quando a condição é satisfeita
- `signal`: acorda um agente e indica que a condição se verifica
- `signalAll`: acorda todos os agentes
- quem re-adquire o lock só procede se a condição é satisfeita

monitor

```
monitor Count {
int i;
condition notNull with (i > 0);
void inc() {
i++;
 // alguem pode usar dec()
 signal(notNull);
}

void dec() {
waitForCondition(notNull);
 i--;
}
}
```

Processos para P_1 e P_2

- apenas modelar a entrada e saída da zona crítica
- supomos que não podem terminar na zona crítica ou ficar lá para sempre
- para representar o ciclo **while** para P_1 temos um estado P_{11} tal que
  - ler os valores de b_2 e k
  - esperar se $b_2 = \textbf{true} \wedge k = 2$
  - mudar de estado, P_{12}
  - em P_{12} entrar e sair da zona critica

Processos para P_1 e P_2

- apenas modelar a entrada e saída da zona crítica
- supomos que não podem terminar na zona crítica ou ficar lá para sempre
- para representar o ciclo **while** para P_1 temos um estado P_{11} tal que
  - ler os valores de b_2 e k
  - esperar se $b_2 = \textbf{true} \wedge k = 2$
  - mudar de estado, P_{12}
  - em P_{12} entrar e sair da zona critica
  - mas como avaliar $b_j \wedge k = j$?

Processos para P_1 e P_2

- apenas modelar a entrada e saída da zona crítica
- supomos que não podem terminar na zona crítica ou ficar lá para sempre
- para representar o ciclo **while** para P_1 temos um estado P_{11} tal que
  - ler os valores de b_2 e k
  - esperar se $b_2 = \textbf{true} \wedge k = 2$
  - mudar de estado, P_{12}
  - em P_{12} entrar e sair da zona critica
  - mas como avaliar $b_j \wedge k = j$?
  - supomos da esquerda para a direita e a segunda não é avaliada se a primeira for falsa

Processos para P_1 e P_2 em CCS

$$
\begin{aligned}
P_1 &:= b1wt!.kw2!.P_{11} \\
P_{11} &:= b2rf?.P_{12} + b2rt?.(kr2?.P_{11} + kr1?.P_{12} \\
P_{12} &:= enter1.exit1.b1wf!.P_1
\end{aligned}
$$

Processos para P_1 e P_2 em CCS

$$
\begin{aligned}
P_1 &:= b1wt!.kw2!.P_{11} \\
P_{11} &:= b2rf?.P_{12} + b2rt?.(kr2?.P_{11} + kr1?.P_{12} \\
P_{12} &:= enter1.exit1.b1wf!.P_1
\end{aligned}
$$

$$
\begin{aligned}
P_2 &:= b2wt!.kw1!.P_{21} \\
P_{21} &:= b1rf?.P_{22} + b1rt?.(kr1?.P_{21} + kr2?.P_{22} \\
P_{22} &:= enter2.exit2.b3wf!.P_2
\end{aligned}
$$

Se $k = 1$ no inicio

$$Peterson \quad := \quad (P_1|P_2|B_{1f}|B_{2f}|K_1)\backslash L$$

onde L são todas as ações excepto as de entrada e saída.

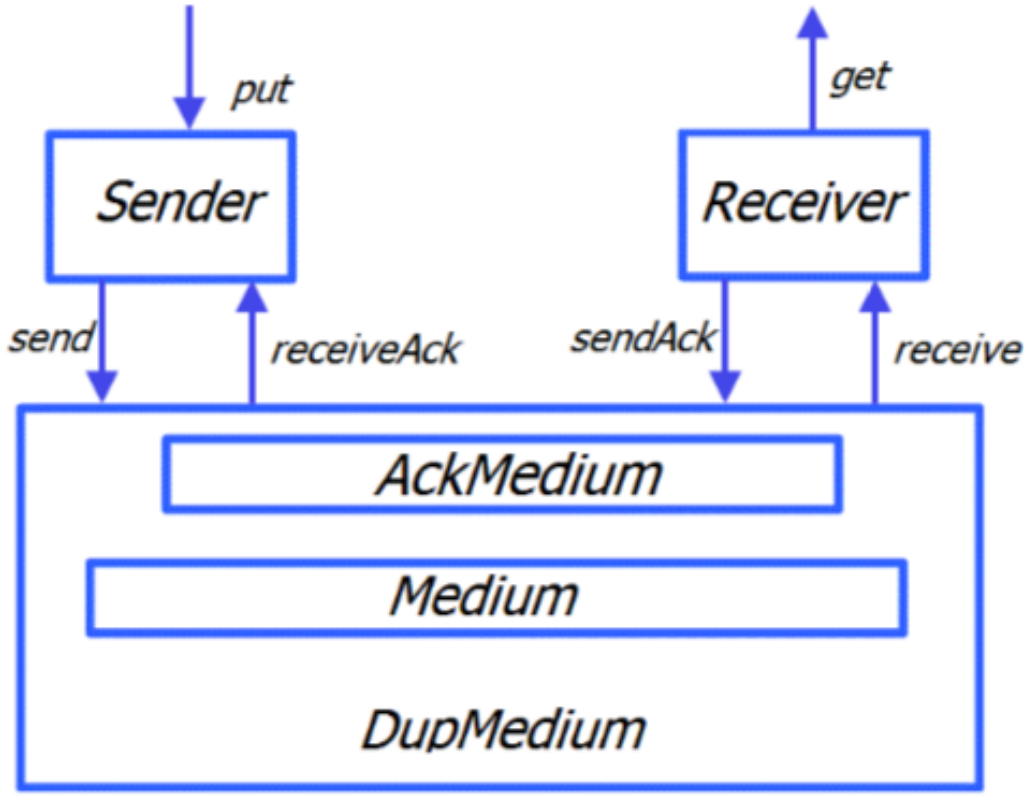
Exclusão mutua em CCS

Se se modelar a entrada e a saída da zona crítica fica

$$MutexSpec \quad := \quad enter1.exit1.MutexSpec + enter2.exit2.MutexSpec$$

Será que $MutexSpec \approx Peterson$?

Exemplo de um Protocolo com erro no meio –Pseuco



Exemplo de um Protocolo com erro no meio –Pseuco

$$
\begin{aligned}
\mathit{Sender} &:= \mathit{put}?.\mathit{send}!.\mathit{Sending} \\
\mathit{Sending} &:= \mathit{receiveAck}?.\mathit{Sender} + \mathit{receiveNAck}?.\mathit{send}!.\mathit{Sending} \\
\mathit{Receiver} &:= \mathit{receive}?.\mathit{get}?.\mathit{sendAck}!.\mathit{Receiver} + \\
&\quad \mathit{gargled}?.\mathit{sendNAck}!.\mathit{Receiver} \\
\mathit{Medium} &:= \mathit{send}?.(\mathit{receive}!.\mathit{Medium} + i.\mathit{garbled}!.\mathit{Medium}) \\
\mathit{AckMedium} &:= \mathit{sendAck}?.\mathit{receiveAck}!.\mathit{AckMedium} + \\
&\quad \mathit{sendNAck}?.\mathit{receivedNAck}!.\mathit{AckMedium} \\
\mathit{DupMedium} &:= \mathit{Medium} | \mathit{AckMedium} \\
\mathit{Protocol} &:= (\mathit{Sender} \mid \mathit{Receiver} \mid \mathit{DupMedium}) \backslash \\
&\quad \{\mathit{send}, \mathit{receive}, \mathit{sendAck}, \mathit{receiveAck}, \\
&\quad \mathit{receiveNAck}, \mathit{sendNAck}, \mathit{garbled}\}
\end{aligned}
$$

$$
\begin{aligned}
Sender &:= put?x.send!x.Sending[x] \\
Sending[x] &:= receiveAck?.Sender + receiveNAck?.send!x.Sending[x \\
Receiver &:= receive?x.get?x.sendAck!.Receiver + \\
&\quad gargled?.sendNAck!.Receiver \\
Medium &:= send?x.(receive!x.Medium + i.garbled!.Medium) \\
AckMedium &:= sendAck?.receiveAck!.AckMedium + \\
&\quad sendNAck?.receivedNAck!.AckMedium \\
DupMedium &:= Medium|AckMedium \\
Protocol &:= (Sender \mid Receiver \mid DupMedium)\backslash \\
&\quad \{send, receive, sendAck, receiveAck, \\
&\quad receiveNAck, sendNAck, garbled\}
\end{aligned}
$$

$$Protocol|put!1.put!2.put!3.put!4.0\backslash\{put\}$$

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)
- ou usar variáveis

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)
- ou usar variáveis
- $a!x$: saída do valor guardado em x no canal a (enviar)

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)
- ou usar variáveis
- $a!x$: saída do valor guardado em x no canal a (enviar)
- $a?x$: entrada de um valor que se guarda em x pelo canal a (receber)

CCS_{vp} com passagem de valor

- $a!v$: saída do valor v no canal a (enviar)
- $a?v$: entrada do valor v pelo canal a (receber)
- ou usar variáveis
- $a!x$: saída do valor guardado em x no canal a (enviar)
- $a?x$: entrada de um valor que se guarda em x pelo canal a (receber)
- e os nomes podem ter variáveis como parametros permitindo assim enviar e receber valores ($A[x, y]$)

CCS_{vp} com passagem de valor

Sendo $\mathbb{V}$ um conjunto de valores e $\mathbb{K}$ um conjunto de canais temos

CCS_{vp} com passagem de valor

Sendo $\mathbb{V}$ um conjunto de valores e $\mathbb{K}$ um conjunto de canais temos

$$
\begin{aligned}
A^{!} &= \{a!v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a! \mid a \in \mathbb{K}\}, \\
A^{?} &= \{a?v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a? \mid a \in \mathbb{K}\}, \\
Com &= A^{!} \cup A^{?} \\
Act &= Com \cup \{\tau\}
\end{aligned}
$$

CCS_{vp} com passagem de valor

Sendo $\mathbb{V}$ um conjunto de valores e $\mathbb{K}$ um conjunto de canais temos

$$\begin{aligned} A^{!} &= \{a!v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a! \mid a \in \mathbb{K}\}, \\ A^{?} &= \{a?v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a? \mid a \in \mathbb{K}\}, \\ Com &= A^{!} \cup A^{?} \\ Act &= Com \cup \{\tau\} \end{aligned}$$

$$\begin{aligned} P &::= 0 \mid X[r_1, \ldots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P\backslash H \\ \chi &::= \tau \mid a! \mid a? \mid a!v \mid a?v \mid a!x \mid a?x \end{aligned}$$

onde $X \in Var$, $x \in D$, $r_i \in D \cup \mathbb{V} \cup \mathbb{K}$

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$\begin{aligned}
(a!y.P)\{v/x\} &= a!y.P\{v/x\} \quad \text{se } y \neq x \\
(a!x.P)\{v/x\} &= a!v.P\{v/x\} \\
(a?y.P)\{v/x\} &= a?y.(P\{v/x\} \quad \text{se } y \neq x \\
(a?x.P)\{v/x\} &= a?x.P \\
X[x]\{v/x\} &= X[v] \\
X[y]\{v/x\} &= X[y] \quad \text{se } y \neq x
\end{aligned}$$

Para as restantes expressões é passado para as subexpressões.

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$\begin{aligned}
(a!y.P)\{v/x\} &= a!y.P\{v/x\} \quad \text{se } y \neq x \\
(a!x.P)\{v/x\} &= a!v.P\{v/x\} \\
(a?y.P)\{v/x\} &= a?y.(P\{v/x\} \quad \text{se } y \neq x \\
(a?x.P)\{v/x\} &= a?x.P \\
X[x]\{v/x\} &= X[v] \\
X[y]\{v/x\} &= X[y] \quad \text{se } y \neq x
\end{aligned}$$

Para as restantes expressões é passado para as subexpressões.
$put?x : 0..9.send!x.0$

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$\text{Rec}\ \frac{P\{v_1/r_1, \ldots, v_n/r_n\} \xrightarrow{\alpha} P' \qquad \Gamma(X[r_1, \ldots, r_n]) = P}{X[r_1, \ldots, r_n] \xrightarrow{\alpha} P'}$$

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde $P\{v/x\}$ é P onde x é substituito por v

$$\text{Rec}\ \frac{P\{v_1/r_1, \ldots, v_n/r_n\} \xrightarrow{\alpha} P' \qquad \Gamma(X[r_1, \ldots, r_n]) = P}{X[r_1, \ldots, r_n] \xrightarrow{\alpha} P'}$$

$$\begin{aligned} send!y.Sending[x]\{3/x, 5/y\} &= send!5.Sending[3] \\ send!y.Sending[x]\{3/x, 5/y, receice/send\} &= receive!5.Sending[3] \end{aligned}$$

when: Condicional bloqueador

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$$B[x] \quad := \quad when(x < 4)put?.B[x+1] + when(x > 0)get?.B[x-1]$$

$B[0]$ ou $B[5]$ o que fazem?

when: Condicional bloqueador

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$$B[x] \quad := \quad when(x < 4)put?.B[x+1] + when(x > 0)get?.B[x-1]$$

$B[0]$ ou $B[5]$ o que fazem?

$$\begin{array}{rcl} send!(x+y)3/x, 5/y & = & sent!(8) \\ 3+5 & \Downarrow & 8 \end{array}$$

when: Condicional bloqueador

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$$B[x] \quad := \quad when(x < 4)put?.B[x+1] + when(x > 0)get?.B[x-1]$$

$B[0]$ ou $B[5]$ o que fazem?

$$\begin{array}{rcl} send!(x+y)3/x, 5/y & = & sent!(8) \\ 3+5 & \Downarrow & 8 \end{array}$$

- $when(b)P$ se b é verdade comporta-se como P senão bloqueia.
- avaliação de expressões $e \Downarrow z$: a expressão e avalia para z

when: Condicional bloqueador

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$$B[x] \quad := \quad when(x < 4)put?.B[x+1] + when(x > 0)get?.B[x-1]$$

$B[0]$ ou $B[5]$ o que fazem?

$$\begin{array}{rcl} send!(x+y)3/x, 5/y & = & sent!(8) \\ 3+5 & \Downarrow & 8 \end{array}$$

- $when(b)P$ se b é verdade comporta-se como P senão bloqueia.
- avaliação de expressões $e \Downarrow z$: a expressão e avalia para z

$$P \quad ::= \quad 0 \mid X[r_1, \ldots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P\backslash H \mid when(b)P$$

when: Condicional bloqueador

Supomos $\mathbb{V} = \mathbb{Z}$ (CCS^{Z}_{vp})

$B[x] \quad := \quad when(x < 4)put?.B[x+1] + when(x > 0)get?.B[x-1]$

$B[0]$ ou $B[5]$ o que fazem?

$$send!(x+y)3/x, 5/y \quad = \quad sent!(8)$$
$$3+5 \quad \Downarrow \quad 8$$

- $when(b)P$ se b é verdade comporta-se como P senão bloqueia.
- avaliação de expressões $e \Downarrow z$: a expressão e avalia para z

$$P \quad ::= \quad 0 \mid X[r_1, \ldots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P\backslash H \mid when(b)P$$

$$IterMult[z, x, y] \quad := \quad when(x > 0)i.IterMult[z+y, x-1, y]$$
$$+when(x == 0)println!z.0$$

$IterMult[0, 3, 7]$

$$\text{Output}\ \frac{e \Downarrow z}{a!e.P \xrightarrow{a!.z} P}$$

$$\text{Output}\ \frac{e \Downarrow z}{a!e.P \xrightarrow{a!.z} P}$$

$$\text{Valor}\ \frac{e \Downarrow z}{\alpha?e.P \xrightarrow{a?z} P}$$

$$\text{Pref}\ \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input}\ \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

$$\text{Rec}\ \frac{P\{v_1/r_1, \ldots, v_n/r_n\} \xrightarrow{\alpha} P' \qquad \Gamma(X[r_1, \ldots, r_n]) = P}{X[r_1, \ldots, r_n] \xrightarrow{\alpha} P'}$$

$$\text{Output}\ \frac{e \Downarrow z}{a!e.P \xrightarrow{a!.z} P}$$

$$\text{Valor}\ \frac{e \Downarrow z}{\alpha?e.P \xrightarrow{a?z} P}$$

$$\text{cond}\ \frac{P \xrightarrow{\alpha} P' \qquad b \Downarrow True}{when(b)P \xrightarrow{\alpha} P'}$$

Células de Memória

$$Cell[rd, wr, x] \quad := \quad rd!x.Cell[rd, wr, x] + wr?y.Cell[rd, wr, y]$$

Células de Memória

$$Cell[rd, wr, x] \quad := \quad rd!x.Cell[rd, wr, x] + wr?y.Cell[rd, wr, y]$$

- $Cell[rd, wr, 5]$
- $Cell[rdA, wrA, 0] | Cell[redB, wrB, 0]$

$$Cell[rd, wr, x] := rd!x.Cell[rd, wr, x] + wr?y.Cell[rd, wr, y]$$

- $Cell[rd, wr, 5]$
- $Cell[rdA, wrA, 0]|Cell[redB, wrB, 0]$

$$\begin{aligned}
Cells &:= Cell[rdA, wrA, 0]|Cell[redB, wrB, 0] \\
Serve &:= mult?.rdA?x : R.rdB?y : R.IterMult[0, x, y] \\
IterMult[z, x, y] &:= when(x > 0)i.IterMult[z + y, x - 1, y] \\
&\quad + when(x == 0)println!z.Serve \\
Use &:= wrA!7.wrB!5.mult!.0
\end{aligned}$$

$(Cells|Serve|Use)\backslash rdA, wrA, rdB, wrB, mult$ Qual o resultado?

Factorial...como sempre

$$\begin{aligned} Fac[n, j] \quad := \quad & when(j > 0)i.Fac[n * j, j - 1] \\ & + when(j == 0)println!n.0 \end{aligned}$$

$Fac[1, 5]$

Factorial...como sempre

$$\begin{aligned} Fac[n,j] &:= when(j>0)i.Fac[n*j, j-1] \\ &\quad + when(j==0)println!n.0 \end{aligned}$$

$Fac[1,5]$

$$\begin{aligned} Fak &:= rdJ?j : R.(when(j>0)rdN?n.wrN!(n*j).wrJ!(j-1 \\ Cell[v,rd,wr] &:= rd!v.Cell[v,rd,wr] + wr?x : R.Cell[x,rd,wr] \\ Cells &:= Cell[0,rdN,wrN] | Cell[0,rdJ,wrJ] \end{aligned}$$

$(wrN!1.wrJ!5.Fak | Cells) \backslash rdN, wrN, rdJ, wrJ$

O CCS_{vp} pode ser embebido no CCS

..logo é só "syntatic sugar"...

CCS_{vp}	CCS
$a!v.P$	$a_v!.P$
$a?x.P$	$\sum_{v \in \mathbb{V}} a_v?.P\{v/x\}$
$X[u_1, \ldots, u_n]$	$X_{u_1,\ldots,u_n}$

Isto é basta usar ações e nomes indexados (podendo ser considerados conjuntos de indices infinitos ($\mathbb{V}$ ou D)

PseuCo: Linguagem de programação concorrente

- Semântica formal traduz-se para CCS e é aí interpretada
- Tem uma semântica executável que é implementada num compilador para Java
- Processos são agentes havendo sempre o `mainAgent`

```
mainAgent { int n, j;
n = 1;
for (j = 5; j > 0 ; j--){
n = n*j;
}
println (" O factorial de 5 e " + n + ".");
}
```

```
mainAgent { int n, j;
n = 1;
for (j = 5; j > 0 ; j--){
n = n*j;
}
println (" O factorial de 5 e " + n + ".");
}
```

$$
\begin{array}{rcl}
MainAgent[a] & := & MainAgent_2[a, 5, 1] \\
MainAgent_1[a, \$j, \$n] & := & MainAgent_2[a, \$j - 1, \$n] \\
MainAgent_2[a, \$j, \$n] & := & when(!(\$j > 0))MainAgent_3[a, \$j, \$n] \\
 & & +when(\$j > 0)i.MainAgent_1[a, \$j, \$n * \$j] \\
MainAgent_3[a, \$j, \$n] & := & println!(" O\ factorial\ de\ 5\ é\ "^{\$}n".").0
\end{array}
$$

$MainAgent[1]\backslash *, println, exception$

Tipos de comunicação entre Agentes

- Por variáveis partilhadas: memória partilhada
- Síncrona via mensagens: canais de capacidade 0
- Assíncrona via mensagens: canais de capacidade > 0 (buffers)

```
void factorial(int z, intchan c) {
    int j, n=1;
    for (j = z; j > 0 ; j--) {
        n= n*j;  }
    c <! n; }
mainAgent {
    intchan cc;
    int mid, fin;
    agent a1 = start(factorial(3, cc));
    println("Agent 1 is working for me.");
    mid = <? cc;
    agent a2 = start(factorial(mid, cc));
    println("Agent 2 is working for me.");
    fin = <? cc;
    println("The factorial of the factorial of
      three is " + fin + ".");}
```

- variáveis podem ser declaradas: locais ou globais
- instruções terminam com ;
- procedimentos tem um valor de retorno ou são de tipo `void`
- condicionais: **if**; operadores $!, ==, \&\&, ||$
- condicionais inline : $n > 5?"mais";"menos"$
- ciclos: **for** e **while**
- estruturas: **struct**
- Em procedimentos
- call-by-value: tipos simples
- call-by-reference: arrays, struct, monitor, mutex e canais

Agentes

- `agent a1=start(<instrucao>)`
- `start(<instrucao);^^I`
- Esperar pela terminação: `join(a1)`

Exemplo

```
int n;
void counter(){
int loop;
for (loop = 0; loop < 5; loop++){
n = n - 1; }
}
mainAgent { n = 10;
agent a1 = start(counter());
agent a2 = start(counter());
join(a1);
join(a2);
println ("The value is "+ n);
}
```

Canais

- `boolchan chan1;` : sincrono
- `intchan7 chan2` : asíncrono
- canais são FIFO: first-in-first-out
- `chan2 <! 7` : envia 7
- `<?chan2` : recebe 7
- `int x=<? chan2`: x fica com 7
- se vazios não enviam
- se se tentar receber de um canal vazio, quem recebe fica bloqueado
- se se tentar enviar para um canal cheio, quem envia fica bloqueado

Select-case

- Vários `case` e um `default`
- não deterministicamente escolhe um que não esteja bloqueado
- `default` nunca está bloquado

```
select {
case chan1 <! a: {
// varias instrucoes}
case b = <? chan2:
// so uma
case <? chan3:
// recebe e esquece
default:
//sempre disponivel
}
```

Memória partilhada

- agentes podem partilhar variáveis
- se simples têm de ser globais
- as estruturas podem ser partilhadas argumentos devido ao call-by-refernce
- Race-condition/Data-race: quando um agente pode escrever numa variável que outro está a let
- Data-race: programas são incorrectos pelo que sem de evitar isso
- Usar **lock** e **monitor** para garantir a exclusão mútua.

lock

- Permitem coordenar o acesso a variáveis evitando o acesso concorrente e assim o data race.
- `lock` : bloqueia
- `unlock`: liberta

lock

- Permitem coordenar o acesso a variáveis evitando o acesso concorrente e assim o data race.
- `lock` : bloqueia
- `unlock`: liberta

```
lock m;
int i=5;
void dec() {
lock(m);
i--;
unlock(m);
}
```

monitor

- É um struct especial
- tem lock implícito: **lock** é usado antes de qualquer acesso a um método do monitor e **unlock** quando o método retorna.

```
monitor count{
int a=0;
int plusPlus)(){
a++;
return a;
}}
```

```
int n;
lock guard n;
void counter(){
int loop;
for (loop = 0; loop < 5; loop++){
lock(guard n);
n = n -1; unlock(guard n);
} }
mainAgent { n = 10;
agent a1 = start(counter());
agent a2 = start(counter());
join(a1);
join(a2);
println ("The value is " + n);
}
```

- Os monitores podem esperar por certas condições (**condition**)
- `waitForCondition` : só procede quando a condição é satisfeita
- `signal`: acorda um agente e indica que a condição se verifica
- `signalAll`: acorda todos os agentes
- quem re-adquire o lock só procede se a condição é satisfeita

```
monitor Count {
int i;
condition notNull with (i > 0);
void inc() {
i++;
 // alguem pode usar dec()
 signal(notNull);
}

void dec() {
waitForCondition(notNull);
 i--;
}
}
```