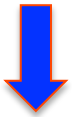
Mapeamento de CSP em CSO

Alexandre Mota
acm@cin.ufpe.br

Abordagem de mapeamento top-down

Arquivo.csp

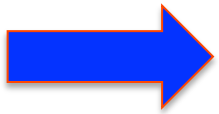
Arquivo.scala

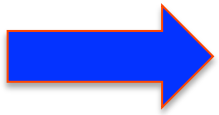


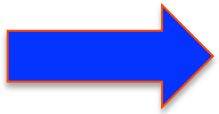
```
package nomePacote
import ox.CSO._
import ox.Format._

object NomeEspec {
   // declarar canais
   // declarar processos
   def main(args: Array[String]) {
       // outras declarações
       comportamentoPrincipal()
       exit
   }
}
```

Declarando canais

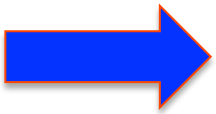
channel ch: Int  val ch = OneOne[Int]

channel ch: T  val ch = OneOne[T]

channel ch: T.{0..Qtd-1}  val ch = OneOne[T](Qtd)

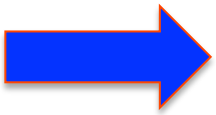
Declarando processos

P = ...



```
def P() = proc {
   ...
}
```

P(S) = ...



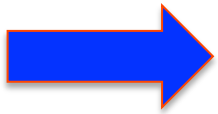
```
def P(S: Tipo) = proc {
   ...
}
```

Mundo aberto vs fechado

- **CSP adota uma solução chamada de Mundo Aberto**
  - Todo canal consegue sincronizar, mesmo que com o ambiente (Nenhum processo explícito)
- **Linguagens de programação adotam o paradigma do Mundo Fechado**
  - Para se ter uma sincronização, deve-se sempre ter um canal de envio (!) para um de recebimento (?)
- **CSO (Scala) adota o paradigma de mundo fechado**

Traduzindo comunicações de entrada

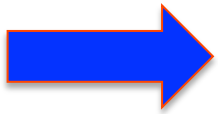
```
channel ch: Int
...
P = ... a?x -> ...
```



```
// Originalmente ch era um canal
...
def P() {
 ...
 val x = Console.readInt
 ...
}
```

Traduzindo comunicações de entrada

```
channel ch: Int
...
P = ... a?x -> ...
```

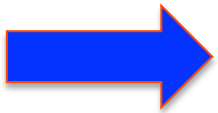


```
...
def P(a: ?[Int]) {
 var x: Int = 0
 ...
 x = a? ;
 ...
}

// No processo que chama P (main)
// declarar
val a = OneOne[Int]
// chamar
P(a)
```

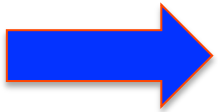
Traduzindo comunicações de saída

```
channel ch: Int
...
P = ... a!v -> ...
```



```
// Originalmente ch era um canal
...
def P() {
 ...
 println("a!${v}")
 ...
}
```

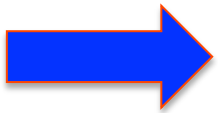
```
channel ch: Int
...
P = ... a!v -> ...
```



```
val a = OneOne[Int] // No processo que
                    // chama P (main)
...
def P(a: ![Int]) {
 ...
 a!v
 ...
}
```

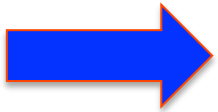
Traduzindo eventos simples (não sincronizam)

```
channel ch
...
P = ... a -> ...
```



```
// Originalmente ch era um canal
...
def P() {
 ...
 println("a")
 ...
}
```

```
channel ch: Int
...
P = ... a -> ...
```



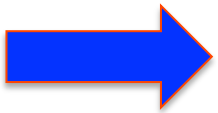
```
// Originalmente ch era um canal
...
def P() {
 ...
 // a
 ...
}
```

Traduzindo eventos simples (sincronizam)

```
channel ch
...
P = ... a -> ...
```

[|{..., a, ...}|]

```
channel ch
...
P = ... a -> ...
```



```
val a = OneOne[Unit]

Em P (ou Q) surge a? (ou a!() )

E

Em Q (ou P) surge a!() (ou a?)
```

Mapeamento dentro de processos

- **O que temos em CSP?**
  - P = pre & a -> P
  - P = a?x:{restricao} -> P
  - P = a!x?y -> P
  - P = (a -> P) [] (b -> P)
  - P = (a -> P) |~| (b -> P)
  - P = a -> Q
  - P = (a -> Q) [] (b -> R)
  - P = Q ||| R
  - P = Q || R
  - P = Q [|{a}|] R
  - ....

CUIDADO
CSO não possui mecanismos naturais para algumas destas construções!

||, [|X|] e |||

- **Em CSO só existe o ||**
  - Isto é, todos os canais de mesmo nome necessitam sincronizar
- **Assim, para obter**
  - P [|X|] Q: deve-se garantir que os canais em X estejam presentes em P e Q (Apenas eles)
  - P ||| Q: Como não há sincronização, estes canais devem assumir uma implementação seguindo o paradigma de mundo fechado (leituras e escritas no console, por exemplo)

Mapeamento de processos

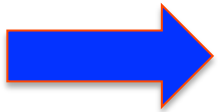
- **pre & P**

- **Isoladamente, a construção pre & P assume a forma de um condicional, onde o else deve resultar em Stop (exceção em CSO)**

```
if(pre) P
else throw new ox.cso.Stop("Stop", new Throwable())
```

Mapeamento de processos

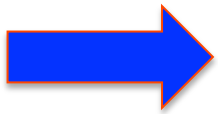
```
g1 & ch1?x -> P1
[] g2 & ch2?y -> P2
...
[] gk & chk!z -> Pk
```



```
alt ( (g1 &&& ch1) =?=> { x => P1 }
    | (g2 &&& ch2) =?=> { y => P2}
    ...
    | (gk &&& chk) =!=> { z } ==> { z => Pk }
)
```

Mapeamento de processos

```
  P1
|~| P2
...
|~| Pk
```



```
import java.util.Random
...
val rand = new Random(System.currentTimeMillis());
val proc_index = rand.nextInt(k)+1;
index match {
 case 1 => // tradução de P1
 case 2 => // tradução de P2
 ...
 case k => // tradução de Pk
}
```

Mapeamento de processos

- **P = a?x:{restrição} -> P**
  - A restrição de um valor de entrada pode ser verificada apenas após a leitura do dado
  - Mas infelizmente já houve a sincronização!!!
    - Poderíamos pensar no rendezvous estendido, mas não seria o mesmo comportamento
  - Então a solução é criar um novo tipo, específico para esta situação e empregar o tradicional:
  - P = a?x -> P

Mapeamento de processos

- **Para o caso de múltiplos dados**
  - P = a!x?y -> …
  - Q = a?x!y -> …
- **Definem-se canais novos, separando as etapas de leitura e escrita…**
- **No 2o caso, usa-se o rendezvous estendido**

Mapeamento de processos

- **Recursão**
  - Opção 1: Usar recursão de Scala normalmente (pode gerar problema de estouro de pilha)
  - Opção 2: Substituir recursão por laço. Neste caso, os parâmetros inicializam variáveis locais e a chamada recursiva simplesmente deixa de existir

Exemplo de tradução

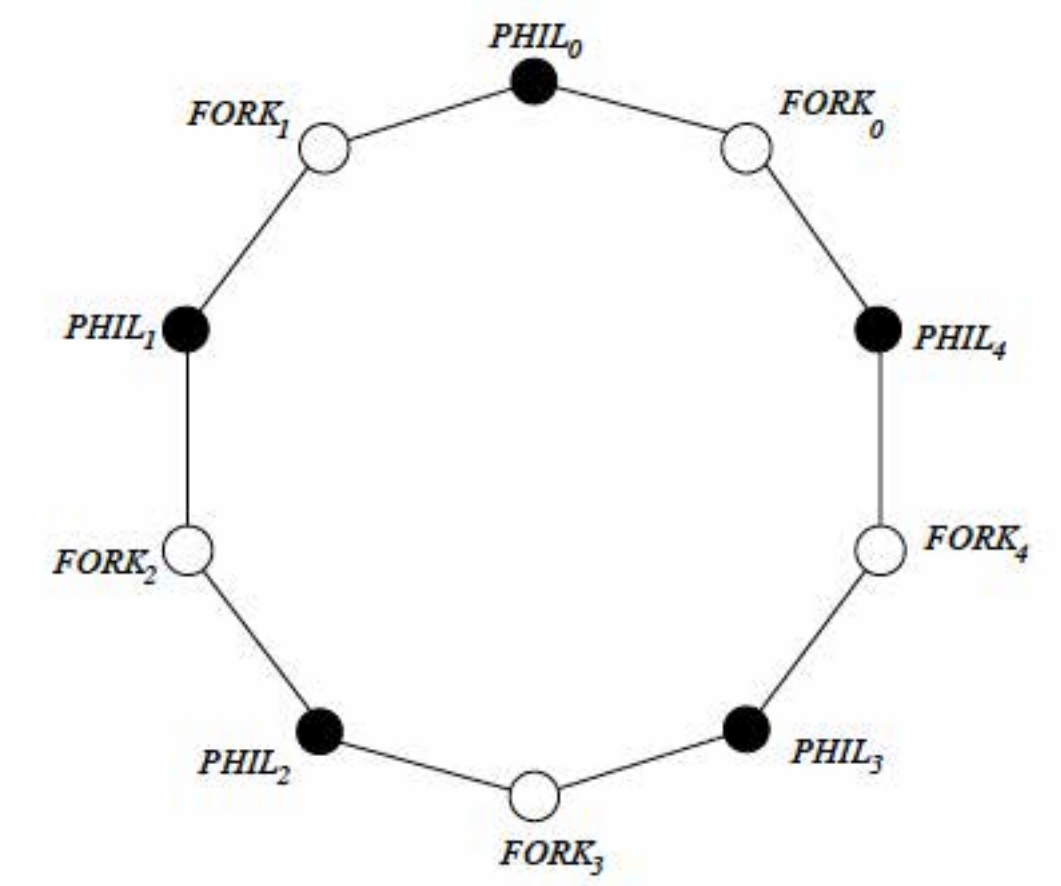
- channel left, right: Int
- copy = left?x -> right!x -> copy

- No eclipse…

Exemplo de tradução

- IDS = {1,2,3,4}
- channel passaCartao, consultaServidor: IDS
- channel libera, barra
- Catraca = passaCartao?id -> consultaServidor!id -> (libera -> Catraca [] barra -> Catraca)
- Servidor = consultaServidor?id -> (libera -> Servidor |~| barra -> Servidor)
- Sistema = Catraca [|{|consultaServidor, libera, barra|}|] Servidor

Exemplo de tradução (Jantar dos filósofos)



Mapeamento de CSP em CSO

Alexandre Mota
acm@cin.ufpe.br