

Breve introdução à CSO (Communicating Scala Objects)

Alexandre Mota

acm@cin.ufpe.br



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

CIn.ufpe.br

- É uma biblioteca na linguagem de programação Scala
- Notação conveniente da essência de OCCAM
 - OCCAM é uma linguagem de programação concorrente que tem por base a comunicação de processos sequenciais (CSP) processo de álgebra
- Baseada em JCSP, mas de fato uma extensão
- Performance próxima ou melhor a de JCSP e da biblioteca de atores de Scala

Principais elementos de CSO

■ Processos

- Simples
- Em paralelo
- Coleção de processos paralelos

■ Portas e canais

- Declarações e tipos
- Canais síncronos e assíncronos

■ Escolha (**alternation**)

Processos

- Quando um processo é inicializado, todos os threads livres necessários para sua execução são adquiridos no pool de threads de Scala
- Ao terminar, os threads são devolvidos ao pool
- Como Scala usa um modelo de concorrência que evita efeitos colaterais (um objeto alterar o estado de outro), coleções são paralelizadas tanto quanto possível
- Apesar de poder ser alterado, a quantidade de threads do pool assume quantidade padrão em função do número de cores da máquina

Processos

- **Processos (p: PROC) são valores**
- **Podem assumir as seguintes formas:**
 - proc { expr }
 - Processo simples (expr deve ser comando, tipo Unit)
 - $p_1 \parallel p_2 \parallel \dots \parallel p_k$
 - Composição paralela de k processos (cada p_i deve ter tipo PROC)
 - \parallel coleção
 - Composição paralela de coleção finita de valores PROC

Coleção de processos

- Um padrão recorrente de uma coleção paralela de processos assume a seguinte forma:
 - $\parallel (\text{for } (i <- 0 \text{ until } k) \text{ yield } p(i))$
- Que é equivalente a escrever
 - $p(0) \parallel p(1) \parallel \dots \parallel p(k-1)$

Inicializando e executando processos

- **Se p é um processo, então avaliar a expressão p() executa o processo**
- **Os seguintes casos são distintos:**
 - p é proc { expr }
 - p() faz { expr } ser avaliada na thread atual
 - O processo como um todo termina quando a avaliação de { expr } termina ou lança uma exceção (não detectada)
 - O comportamento da expressão p() não pode distinguir-se do comportamento da expressão { expr }

Inicializando e executando processos

■ $p \in p_1 \parallel p_2 \parallel \dots \parallel p_k$

- $p()$ faz todos os processos $p_1 \parallel p_2 \parallel \dots \parallel p_k$ executarem concorrentemente
- Todos os processos exceto um (este executa na thread atual) são executados em nova thread
- O processo como um todo termina quando todos os componentes p_i terminarem. Se um ou mais p_i 's terminarem lançando exceção não capturada, estas exceções são empacotadas em uma ParException que é re-lançada, a menos que elas sejam todas subtipos de cso.Stop; neste caso uma única cso.Stop é lançada

Portas e canais

- Portas em CSO são parametrizadas em geral
- Usam-se as abreviações ?[T] e ![T] (que correspondem a InPort[T] e OutPort[T])
- Chan[T] pode ser InPort[T] ou OutPort[T]
- Os métodos mais importantes são:
 - ! (valor: T)
 - Serve para enviar uma mensagem de tipo T
 - ? (): T
 - Serve para receber mensagem do tipo T
 - ? [U] (body: T => U) : U
 - Método rendezvous estendido

Explicado e exemplificado posteriormente

Portas e canais (Mais frequentes)

■ Canais síncronos

- OneOne[T]
 - Apenas um processo pode fazer ? ou !
- ManyOne[T]
 - Apenas um processo pode fazer ?. Mais de um processo pode fazer !, mas não-deterministicamente
- OneMany[T]
 - Inverso do anterior
- ManyMany[T]
 - Combinacão dos dois anteriores. Não-determinismo para ? e !

■ Buf[T](n) – buffer de capacidade n

Exemplo 1

Composição paralela

```
def producer(i: int, ![T]) : PROC = ...
def consumer(i: int, ?[T]) : PROC = ...

val con = OneOne[T](n)      // an array of n unshared channels

(|| (for (i<-0 until n) yield producer(i, con(i)))
|| (for (i<-0 until n) yield consumer(i, con(i)))
)||()
```

Canais simples

Coleções de processos

Instanciando vetor de canais

Execução dos processos

Rendezvous estendido

- **Como dito antes, uma escrita (!) termina sincronamente com a terminação da leitura (?)**
- **Uma leitura usando rendezvous estendido**
 - Permite que a computação sobre os dados transferidos seja feita no processo leitor
 - E apenas quando esta computação termina é que a leitura é considerada terminada e o processo escritor é liberado da sincronização
- **A forma usual é**
 - `in ? { bv => body }`
 - Recebe v através de in e aplica a função `{ bv => body }` sobre v

Exemplo: monitorando tráfego entre processos

■ Dado o processo

```
{ val mid = Chan[T]
  producer(mid) || consumer(mid)
}
```

- Suponha que se deseja monitor o tráfego de dados entre producer e consumer
- Uma primeira aproximação seria

```
{ val left , mon, right = Chan[T]
  producer(left)
  || proc { repeat { val v = left ?; mon!v; right!v }
    consumer(right)
    || monitor(mon)
  }
}
```

Exemplo: monitorando tráfego entre processos

■ De

```
{ val left , mon, right = Chan[T]
  ( producer(left)
  || proc { repeat { val v = left ?; mon!v ; right !v }
  || consumer(right)
  || monitor(mon)
  )
}
```

- Temos o problema de que tão logo **left ?** Seja executado, **producer** está livre para continuar
- Solução seria...

```
{ left ? { v => {mon!v; right !v} } }
```

Exemplo: monitorando tráfego entre processos

■ Mas então

```
{ left ? { v => {mon!v; right!v} } }
```

■ O rendezvous só termina, quando a expressão { mon ! v ; right ! v } terminar

- Há potencial para deadlock, devido à ordem

■ Para relaxar, pode-se usar

```
{ left ? { v => {(proc{mon!v} || proc{right!v}))()} } }
```

Exemplo: monitorando tráfego entre processos

- Pode-se generalizar a solução anterior definido o seguinte componente

```
def tap[T](in: ?[T], out: ![T], mon: ![T]) =  
  proc  
    { repeat { in ? { v => {(proc{mon!v} || proc{out!v})()} } } }
```

Outro exemplo interessante

```
def copyToNet[T](in: ?[T], net: ![T], ack: ?[Unit]) =  
  proc { repeat { in ? { v => { net!v; ack? } } } }
```

Um evento
transformado em
canal (Unit)

```
def copyFromNet[T](net: ?[T], ack: ![Unit], out: ![T]) =  
  proc { repeat { out!(net?); ack!() } }
```

Escolha (alternation)

- A forma mais simples de escolha consiste em uma coleção de eventos guardados

```
alt ( (guard1 && port1) => { bv1 => cmd1 }
    |
    |
    | ... 
    | (guardn && portn) => { bvn => cmdn }
    )
```

- Um evento da forma $(g \&\& p) =?> \{ bv \Rightarrow cmd \}$
 - É dito *habilitado* se p está aberta e g for true
 - É dito *pronto* se p estiver pronta para leitura
 - É *disparado* pela leitura de p, associando o valor lido a bv e executando cmd

Escolha (alternation)

- **A execução de uma alt procede, em princípio, em fases seguintes:**
 - Todas as guardas dos eventos são avaliadas e, em seguida
 - O thread atual aguarda até que (pelo menos um) caso habilitado esteja pronto e, então
 - Um dos eventos prontos é escolhido e disparado
- **Se não houver eventos habilitados após a fase 1, ou se todos os canais associados com as portas fecharem enquanto espera na fase 2, então a exceção Abort (que é também uma forma de exceção Stop) é levantada**

Escolha (alternation)

- Um construtor especial sobre escolhas é o **serve**
- Se **evs** é uma coleção de eventos guardados, então, **serve (evs)** executa as fases anteriores repetidamente (até que uma exceção Stop seja lançada)
 - Mas as escolhas feitas na fase 3 são feitas em esquema de rodízio (para evitar repetição)

Exemplo com escolha

```
def tagger[T](l: ?[T], r: ?[T], out: ![(int, T)]) =  
proc  
{ var diff = 0  
  serve ( ((!r.open || diff < 5) &&& l) => { x => out!(0, x); diff+=1 }  
         | ((!l.open || diff > -5) &&& r) => { x => out!(1, x); diff-=1 }  
         )  
  ( proc {l.closein} || proc {r.closein} || proc {out.closeout} )()  
}
```

Escolha guardada usando canais de saída

■ Em 2008, a forma

- (guard&&&port) =!=> {expression} ==> {bv => cmd}

■ Foi adicionada à CSO

■ Seu funcionamento é:

- Quando a porta estiver *pronta* para comunicar, então a forma completa estará *pronta* (estado)
- É *disparada* avaliando a expressão e escrevendo o valor resultante (bv) na porta
 - O valor resultante também pode ser usado em cmd

Escolha guardada usando canais de saída

- **O valor bv pode ser omitido se não for usado em cmd**
 - (guard&&&port) =!=> { expression } ==> { cmd }
- **E a última forma possível é**
 - (guard&&&port) =!=> { expression } ==> { cmd }
- **Caso cmd não necessite ser usado**

Exemplo

```
def taggedMerge[T](l: ?[T], r: ?[T], out: ![(int, T)]) =  
proc  
{ var seqn = 0 // sequence number  
  var nbuf = 0 // number buffered  
  val q      = scala.collection.mutable.Queue[(Int, Int, T)]  
  serve ( (nbuf<20 && l)  => { x => q.enqueue((seqn+=1, 0, x)); nbuf+=1; }  
    | (nbuf<20 && r)  => { x => q.enqueue((seqn+=1, 1, x)); nbuf+=1; }  
    | (nbuf>0  && out) => { q.dequeue } ==> { nbuf-=1; }  
  )  
  ( proc {l.closein} || proc {r.closein} || proc {out.closeout} )()  
}
```

Referências

- <https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/CSO/cpa2008-cso-2014revision.pdf>
- <https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/CSO/cso-doc-scala2.11.4/#package>

Breve introdução à CSO (Communicating Scala Objects)

Alexandre Mota

acm@cin.ufpe.br



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

CIn.ufpe.br

Exemplo 2

```
def producer(i: int, ![T]) : PROC = ...
def consumer(i: int, ?[T]) : PROC = ...

def mux[T] (ins: Seq[?[T]],    out: ![(int, T)]) : PROC = ...
def dmux[T](in:  ?[(int, T)], outs: Seq[![T]])   : PROC = ...

val left, right = OneOne[T](n) // 2 arrays of n unshared channels
val mid = OneOne[(int, T)]      // an unshared channel

(|| (for (i<-0 until n) yield producer(i, left(i)))
|| mux(left, mid)
|| dmux(mid, right)
|| || (for (i<-0 until n) yield consumer(i, right(i)))
)()
```

Exemplo 2

```
def mux1[T] (ins: Seq[?[T]], out: !(Int, T)) : PROC =  
{ val mid = ManyOne[Int, T]  
  ( proc { while(true) { out!(mid?) } }  
  || (for (i<-0 until ins.length) yield  
       proc { while(true) {mid!(i, ins(i)?)} })  
  )  
}  
  
def mux2[T] (ins: Seq[?[T]], out: SharedOutPort[Int, T]) : PROC =  
|| (for (i<-0 until ins.length) yield  
    proc { while (true) {out!(i, ins(i)?)} })  
  
def dmux[T](in: ?[(Int, T)], outs: Seq[![T]]) : PROC =  
proc {  
  while (true) { val (n, v) = in?; outs(n)!v }  
}
```