CCS: Calculus of Communicating Systems

Riccardo Renzulli
9 Dicembre 2015

Overview

- Introduzione al CCS
- Sintassi
- Semantica
- Esempio di riduzione
- Interprete del CCS in Haskell
- Vantaggi e svantaggi nell’aver usato Haskell

Introduzione al CCS

Introduzione al CCS

- Il CCS (Calculus of Communicating Systems) fu introdotto da Robin Milner nel 1980.
- Il CCS è un modello matematico (un linguaggio formale) per descrivere processi, principalmente studiato nella programmazione concorrente.
- Un programma CCS, scritto in una sintassi per esprimere le espressioni di comportamento, denota il comportamento di un processo.
- Il CCS è quindi un semplice linguaggio di programmazione concorrente basato sullo scambio esplicito di messaggi.
- In questa presentazione verrà considerata una versione “semplificata” del CCS originale.

Sintassi

Sintassi

- 0 Processo nullo
- $I = P$ Definizione
- $a.P$ Action prefixing con $a \in Actions$
- $P_1 + P_2$ Scelta non deterministica
- $P_1 \mid P_2$ Composizione parallela

Semantica

Semantica

- **Prefixing**

$$a.P \xrightarrow{a} P$$

- **Scelta**

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

Semantica

- **Composizione parallela**

Comunicazione esterna

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}$$

Comunicazione interna

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Semantica

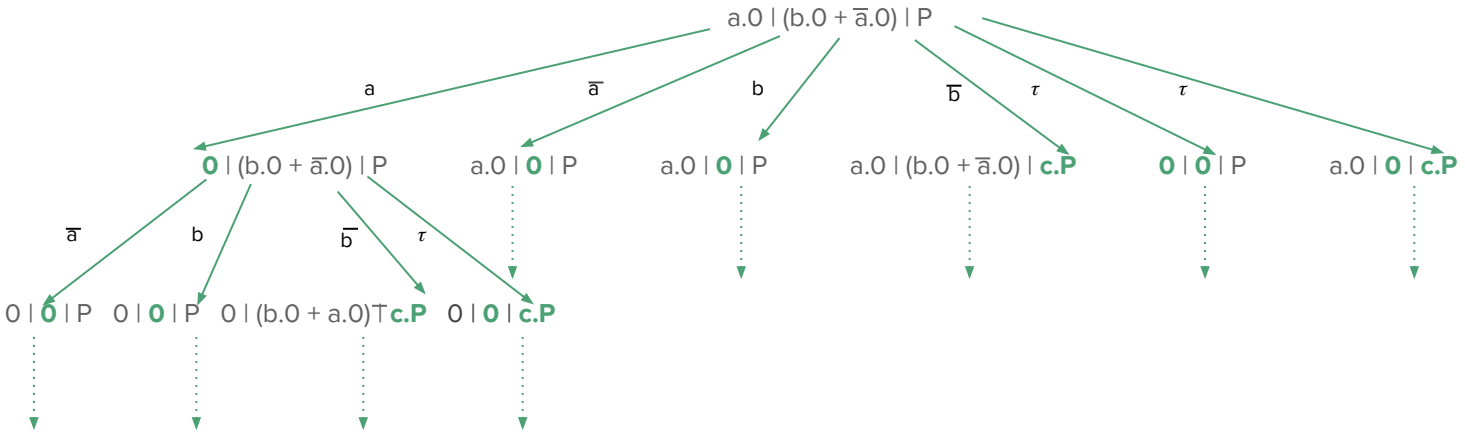
- **Definizione**

$$\frac{P \xrightarrow{a} P' \quad I = P}{I \xrightarrow{a} P'}$$

Esempio di riduzione

Esempio di riduzione

$P = \overline{b}.c.P$

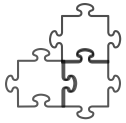


Interprete del CCS in Haskell

(solo comunicazioni interne)

Interprete del CCS in Haskell

Grammatica



Testo - - > Lexer - - > (Tokens) - - > Parser - - > **Interprete** - - > **Albero con passi di riduzione**

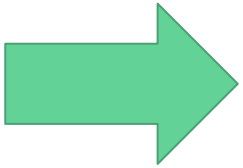
Interprete del CCS in Haskell

Testo - - > Lexer - - > (Tokens) - - > Parser - - > Interprete - - > Albero con passi di riduzione

CS = $\overline{coin}$.coffee.CS

CM = coin.$\overline{coffee}$.CM

CS | CM



```
var CS,CM;
 let CS  = !coin.coffee.CS,
       CM = coin.!coffee.CM;
in CS | CM
```

Interprete del CCS in Haskell

Testo - - > **Lexer - - > (Tokens) - - > Parser** - - > Interprete - - > Albero con passi di riduzione

Per scrivere il parser è stato usato il parser generator “Happy”.

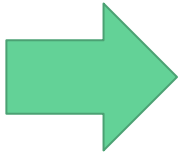
```
Prog     : var Expr ';' Stmt
         | Stmt

Expr     : Expr ',' id
         | id

Stmt     : let Decl ';' in ListProc
         | ListProc

Decl     : Decl ',' id '=' Process
         | id '=' Process

ListProc : ListProc '|' Process
         | Process

Process  : iaction '.' Process
         | oaction '.' Process
         | Process '+' Process
         | '(' Process ')'
         | id
         | '0'
```

La grammatica

Interprete del CCS in Haskell

Testo - - > **Lexer - - > (Tokens) - - > Parser** - - > Interprete - - > Albero con passi di riduzione

P ::= *a*.P | P + Q | P|Q | I | 0



```
data Process = InputAction [Char] Process
             | OutputAction [Char] Process
             | ExtChoice Process Process
             | Par [Process]
             | Id [Char]
             | Nil
```

Interprete del CCS in Haskell

Testo - - > Lexer - - > (Tokens) - - > **Parser** - - > Interprete - - > Albero con passi di riduzione

Il parser restituisce la symbol table e il processo Par [P_1,P_2,...,P_n] dove P_1,P_2,...,P_n sono i processi da eseguire in parallelo.

parser :: [Token] -> ([([Char],Process)],Process)

Esempio: **CS** = !coin.coffee.CS, **CM** = coin.!coffee.CM

([("CS", OutputAction "coin" (InputAction "coffee" (Id "CS"))), ("CM", InputAction "coin" (OutputAction "coffee" (Id "CM")))], Par [Id "CS", Id "CM"])

Interprete del CCS in Haskell

Testo - - > Lexer - - > (Tokens) - - > Parser - - > **Interprete** - - > Albero con passi di riduzione

interpreter :: [([Char],Process)] -> Process -> (Int,Int) -> Int -> Tree (Process, (Int,Int))

Il “cuore” dell’interprete:

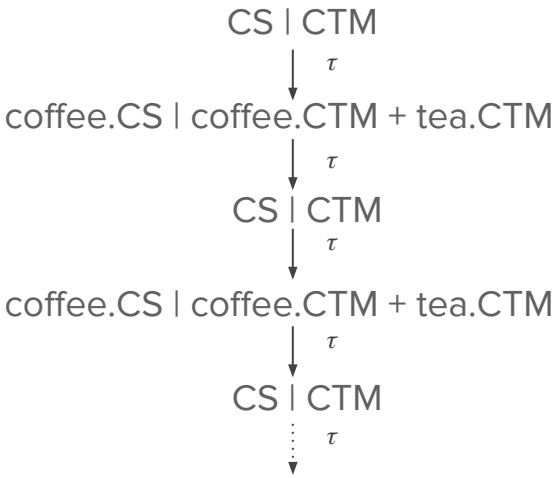
```
evaluate2Proc :: [([Char],Process)] -> (Process,Process) -> [(Process,Process)]
evaluate2Proc symT (InputAction az1 p1,p2)  = case p2 of OutputAction az2 p3 -> if (az1 == az2) then [(p1,p3)] else []
                                                         ExtChoice p3 p4     -> evaluate2Proc symT (InputAction az1 p1,p3) ++ evaluate2Proc symT (InputAction az1 p1,p4)
                                                         Id id1              -> evaluate2Proc symT (InputAction az1 p1,(search symT id1))
                                                         _                   -> []

evaluate2Proc symT (OutputAction az1 p1,p2) = case p2 of InputAction az2 p3  -> if (az1 == az2) then [(p1,p3)] else []
                                                         ExtChoice p3 p4     -> evaluate2Proc symT (OutputAction az1 p1,p3) ++ evaluate2Proc symT (OutputAction az1 p1,p4)
                                                         Id id1              -> evaluate2Proc symT (OutputAction az1 p1,(search symT id1))
                                                         _                   -> []

evaluate2Proc symT (ExtChoice p3 p4,p2)     = case p2 of InputAction az2 p5  -> evaluate2Proc symT (p3,p2) ++ evaluate2Proc symT (p4,p2)
                                                         OutputAction az2 p5 -> evaluate2Proc symT (p3,p2) ++ evaluate2Proc symT (p4,p2)
                                                         Id id1              -> evaluate2Proc symT (ExtChoice p3 p4,(search symT id1))
                                                         ExtChoice p5 p6     -> evaluate2Proc symT (p3,p5) ++ evaluate2Proc symT (p3,p6) ++ evaluate2Proc symT (p4,p5) ++ evaluate2Proc symT (p4,p6)
                                                         _                   -> []

evaluate2Proc symT (Id id1,p2)              = case p2 of Id id2 -> evaluate2Proc symT ((search symT id1),(search symT id2))
                                                         Nil    -> []
                                                         _      -> evaluate2Proc symT ((search symT id1),p2)

evaluate2Proc symT (Nil,p2)                 = []

evaluate2Proc symT (p1,p2)                  = error "Processo sconosciuto"
```

Interprete del CCS in Haskell

Testo - - > Lexer - - > (Tokens) - - > Parser - - > Interprete - - > **Albero con passi di riduzione**

Esempio di parallelismo infinito

CS = !coin.coffee.CS, **CTM** = coin.(!coffee.CTM + !tea.CTM)



Vantaggi e svantaggi nell’aver usato Haskell

VANTAGGI

- Approccio funzionale puro
- Pattern matching
- Programma più pulito

SVANTAGGI

- Nessuno

Grazie per l’attenzione!

riccardo.renzulli@edu.unito.it