Lezione 9 MSC
CCS – introduzione e sintassi

Roberto Gorrieri

Un linguaggio per descrivere modelli

Perché può essere utile avere un tale linguaggio?

- Fornire una supporto linguistico per descrivere succintamente modelli di un sistema, possibilmente in modo composizionale
- Fornire una strumento di prototipazione del modello (se il linguaggio usato è eseguibile)
- Fornire supporto per un'analisi composizionale del modello
- Fornire supporto per un ragionamento equazionale

Supporto linguistico

- Eccetto per sistemi relativamente piccoli, una rappresentazione grafica per mezzo di LTS è laboriosa; in effetti, sistemi reali hanno migliaia (o perfino milioni) di stati, il che rende impossibile definirli e disegnarli.
- Necessità di una rappresentazione testuale (lineare) implicita per mezzo di un termine in un qualche linguaggio per descrivere sistemi concorrenti. (Pensa alle espressioni regolari vs linguaggi regolari)
- Il linguaggio scelto è CCS (Calculus of Communicating Systems)
- Semantica operazionale che associa ad ogni termine del CCS un LTS.
- Teorema di rappresentazione: ogni LTS (a stati finiti) ha un termine CCS che lo rappresenta.

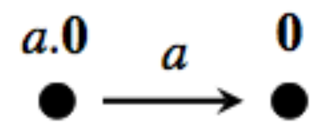
CCS (1): processo nil
ed operatore di prefisso

- Operatore **0** (detta nil , processo vuoto)
- Operatore di prefisso _._, ad esempio a.p (dove a è un'azione e p un processo): e.g., a.0 oppure a.b.0

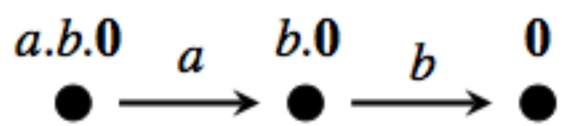
Con 0 e prefisso posso solo costruire processi "lineari" (ogni stato ha al più una transizione in uscita) e "finiti" (ogni cammino è finito)

- Lts per nil

- Lts per a.0



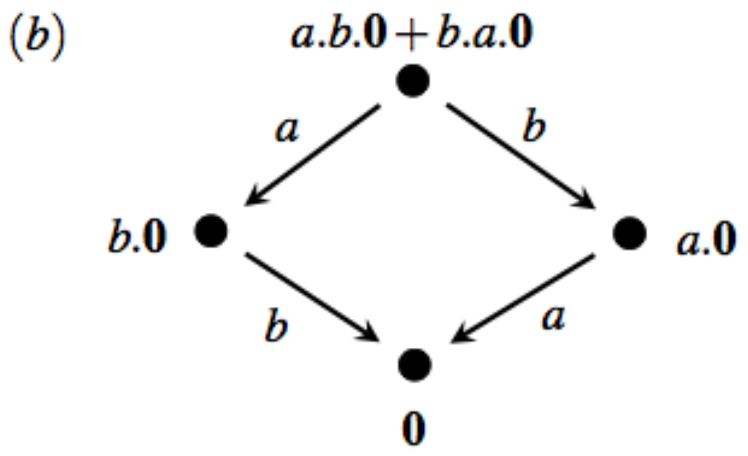
- Lts per a.b.0



CCS (2): Operatore di scelta alternativa +

- Per esprimere opzioni alternative, così come per esprimere nondetermi-nismo, si usa l'operatore di scelta _+_
- p + q può o eseguire un'azione da p (e poi proseguire col residuo di p) oppure eseguire un'azione da q (e poi proseguire col residuo di q): in ogni caso, l'alternativa non selezionata viene scartata.

- Lts per a.b.0 +b.a.0



Esercizio: Come sono fatti gli lts per a.b.0 + a.c.0   e a.(b.0 + c.0)?

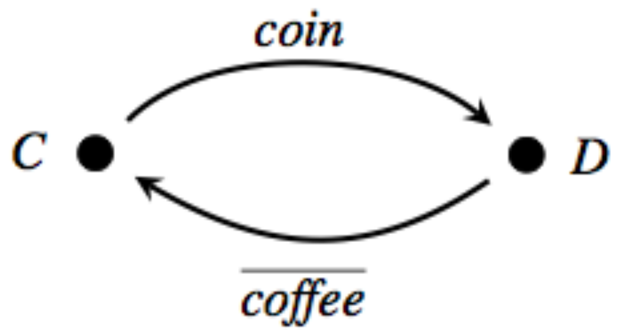
Osserva che nel primo c'è nonde-terminismo su *a* mentre nel secondo opzioni in alternativa (*b* e *c*)

CCS (3): Costanti e loro definizioni

- Costante (nome di processo) A, con associata definizione, permette comportamenti ciclici e ricorsivi (cioè “infiniti”)
- Si possono utilizzare un numero arbitrario di costanti, ma di solito consideriamo “ragionevoli” solo quei termini che usano un numero finito di costanti.

- Lts per A = a.A

- Lts per C = coin.D e D = ‘coffee.C

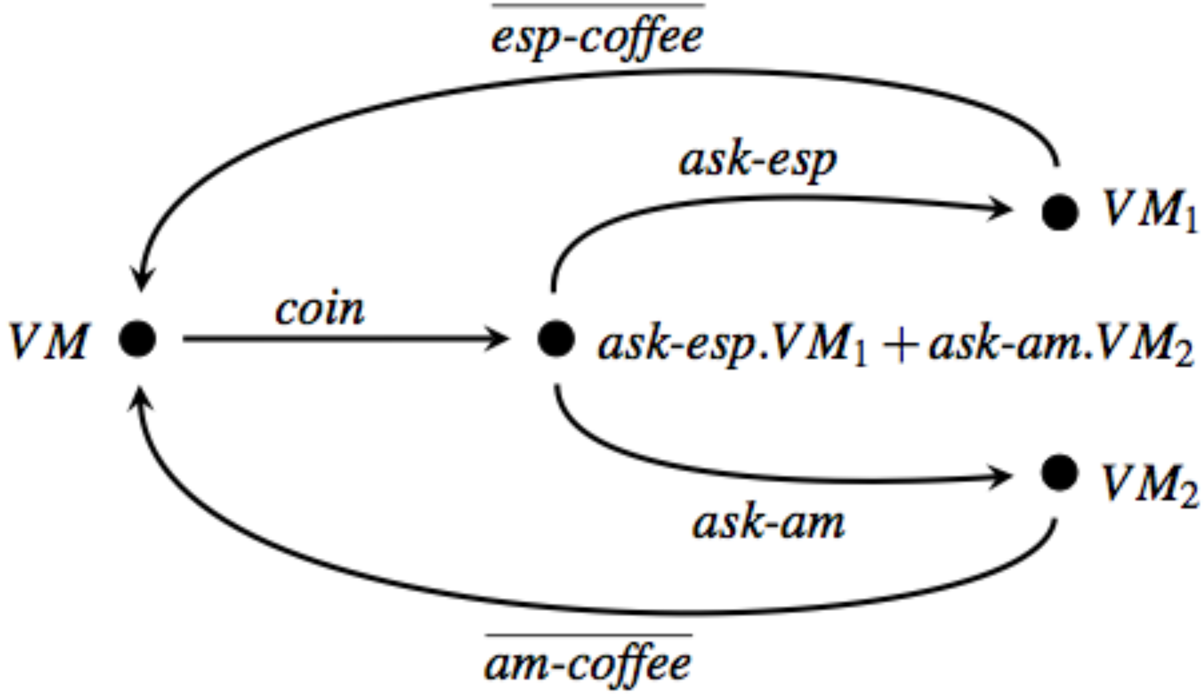


- Lts per C = coin.’coffee.C è isomorfo a quello sopra.

Esempio

$$VM \stackrel{def}{=} coin.(ask\text{-}esp.VM_1 + ask\text{-}am.VM_2)$$

where $VM_1 \stackrel{def}{=} \overline{esp\text{-}coffee}.VM$ and $VM_2 \stackrel{def}{=} \overline{am\text{-}coffee}.VM$.

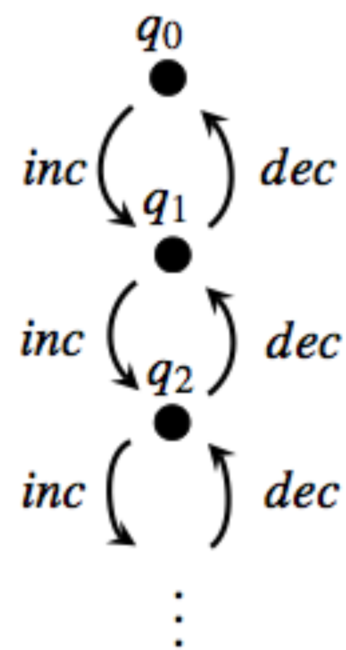


Finite-state CCS

- Con nil, action prefixing, l'operatore di scelta e un numero finito di costanti, possiamo definire qualunque lts a stati finiti!
- Il sottolinguaggio di CCS così ottenuto è detto finite-state CCS.
- Sebbene espressivo, finite-state CCS non è molto utile dal punto di vista di modellazione, perché non permette di modellare i vari sottoprocessi che compongono un sistema complesso (modellazione composizionale).
- Mancano due operatori cruciali: composizione parallela e restrizione.

Cosa si può fare con infinite costanti?

- Se vogliamo definire lts's a stati infiniti, dobbiamo o introdurre infinite costanti, o aggiungere nuovi operatori.
- Esempio: semi-counter
- Ogni stato q_k rappresenta la costante $SCount_k$ per k in N



$$SCount_0 \stackrel{def}{=} inc.SCount_1$$

$$SCount_n \stackrel{def}{=} inc.SCount_{n+1} + dec.SCount_{n-1} \quad n > 0$$

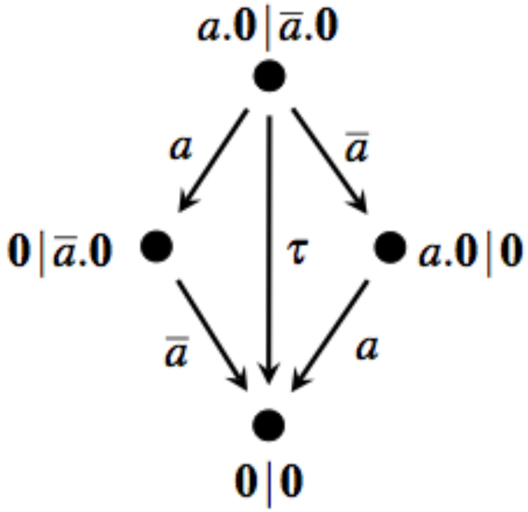
- Ogni lts finitely-branching con infiniti stati può essere rappresentato con infinite costanti (e con somma e prefisso)

Operatori dinamici vs statici

- Oltre agli operatori di prefisso . e di scelta + (detti operatori dinamici perché “svaniscono” mentre la computazione procede), ed oltre all’operatore nil, in CCS esistono due operatori statici: composizione parallela p|q e di restrizione (va)p.
- Se questi ultimi non vengono mai usati all’interno di definizioni di costanti, allora il sottolinguaggio che si ottiene è detto regular CCS, ed ha la caratteristica che i suoi processi sono ancora a stati finiti.
- Quindi regular CCS è molto utile per descrivere in modo composizionale sistemi a stati finiti.

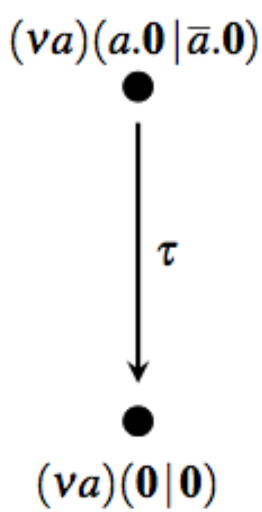
Composizione parallela

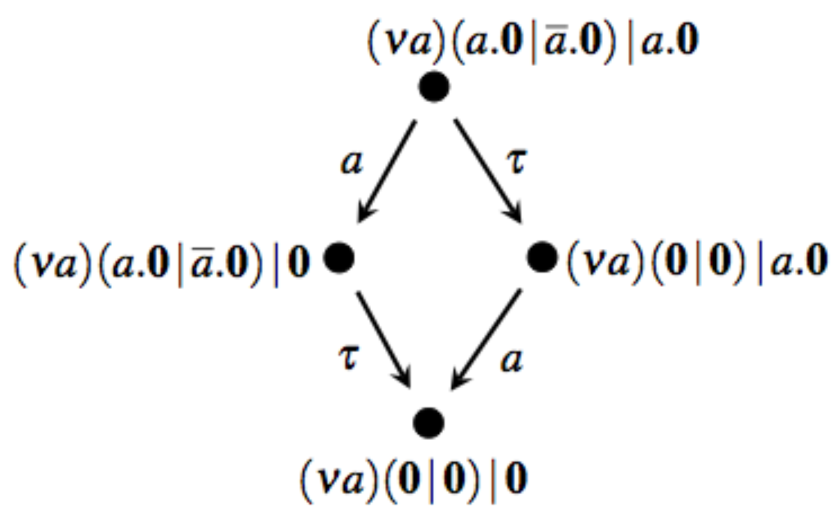
- Due processi indipendenti p1 e p2 possono girare in parallelo quando li componiamo con l'operatore di composizione parallela: p1 | p2.
- Questo significa che i due possono eseguire le loro azioni asincronamente (a velocità relativa impredicibile), o interagire eseguendo sincronamente azioni di input/output complementari (detta handshake synchronization), generando un'azione tau.
- Quindi la sincronizzazione è strettamente binaria!
- Nota che la sincronizzazione non è obbligata: l'azione *a* può anche essere eseguita asincronamente per rappresentare la sua disponibilità ad interagire con l'ambiente esterno.
- Nota che l'operatore non "svanisce", da qui il nome di statico.



Restrizione

- Per rendere privata ad un processo p una certa azione a, si usa l'operatore di restrizione (che è statico).
- (va)p dichiara che l'azione *a* che p potrebbe eseguire non può essere offerta per interazione all'ambiente esterno (cioè ad un eventuale altro processo in parallelo), e che – dentro p – *a* può essere usata solo per sincronizzazioni interne.



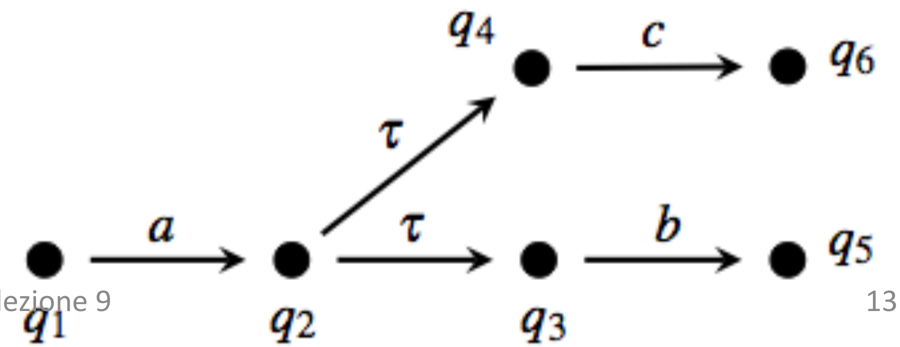


Esempio (1) -- competizione

- Col parallelo e la restrizione, possiamo esprimere, tra le tante cose, anche la competizione tra processi indipendenti per l’uso di risorse condivise, quindi una forma di scelta (interna). Ad esempio:

  (vd)(a.(d.b.0 | d.c.0) | ‘d.0) con ‘d indico il complemento di d

  Nota che c’è un fork dopo a: cioè il processo inizialmente sequenziale a.(d.b.0|d.c.0) eseguendo a attiva due processi concorrenti d.b.0 e d.c.0 in competizione per la risorsa condivisa ‘d.0



Esempio (2) -- sequenzializzazione

- Action prefixing esprime una forma molto banale di sequenzializzazione: "un azione precede un processo".
- Con parallelo e restrizione si può esprimere una forma più generale in cui "un processo precede un altro processo".
- Ad esempio: con (vd)(a.(b.d.0 + c.d.0) | 'd.q) descriviamo un processo in cui q viene attivato solo quando il processo di sinistra ha completato l'esecuzione (con ab o ac).
- Altro esempio: in (vd)((b.d.0 | c.d.0) | 'd.'d.q) il processo q viene attivato solo dopo che i due processi b.d.0 e c.d.0 sono terminati.
- È possibile adattare l'idea sopra per rappresentare un operatore derivato p;q che permette di sequenzializzare p e q

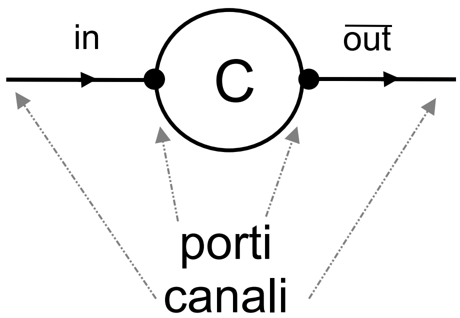
Flow graph – Diagramma dell’architettura del sistema

- In un processo q, gli operatori statici determinano l’architettura di interconnessione del sistema.
- Ogni processo “sequenziale” p viene rappresentato da un cerchio con etichetta “p”.
- Ogni azione di p determina un port su cui è possibile instaurare un canale di comunicazione con port complementari di altri processi sequenziali.
- Se il nome è ristretto tra due processi sequenziali, allora il canale instaurato tra i due è privato.

Flow graph di processo sequenziale

Agente C "buffer ad una posizione"

FLOW-GRAPH



Descrizione architetturale del sistema (connessioni)

in e 'out sono i nomi dei canali di input e output, rispettivamente

$$\left.\begin{array}{lcl} C & = & in.C' \\ C' & = & \text{'}out.C \end{array}\right\}$$

specifica in CCS, che usa 2 costanti, ma il processo è uno solo!

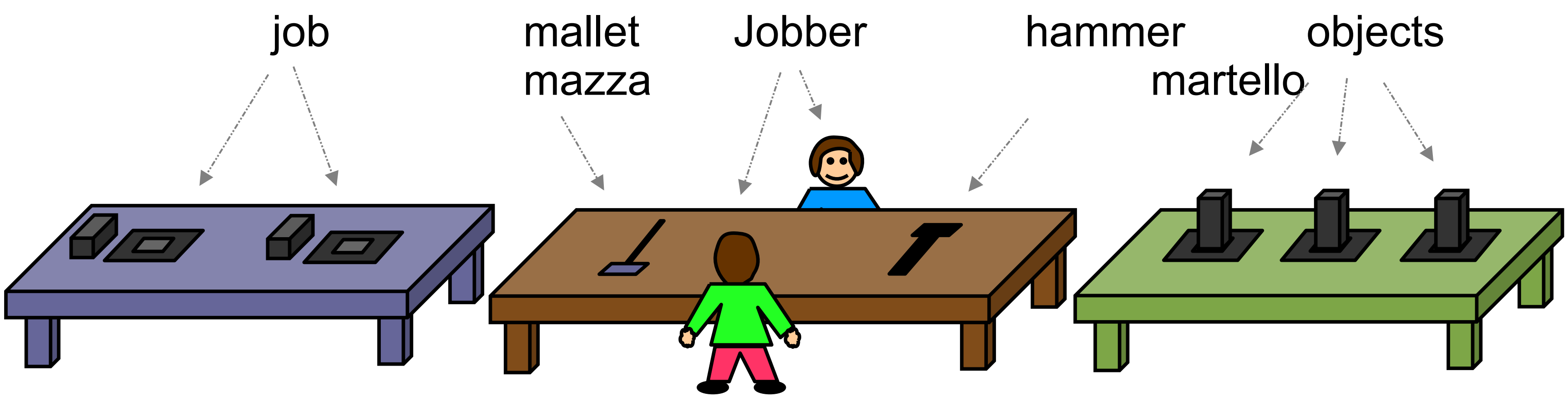
C = in.'out.C

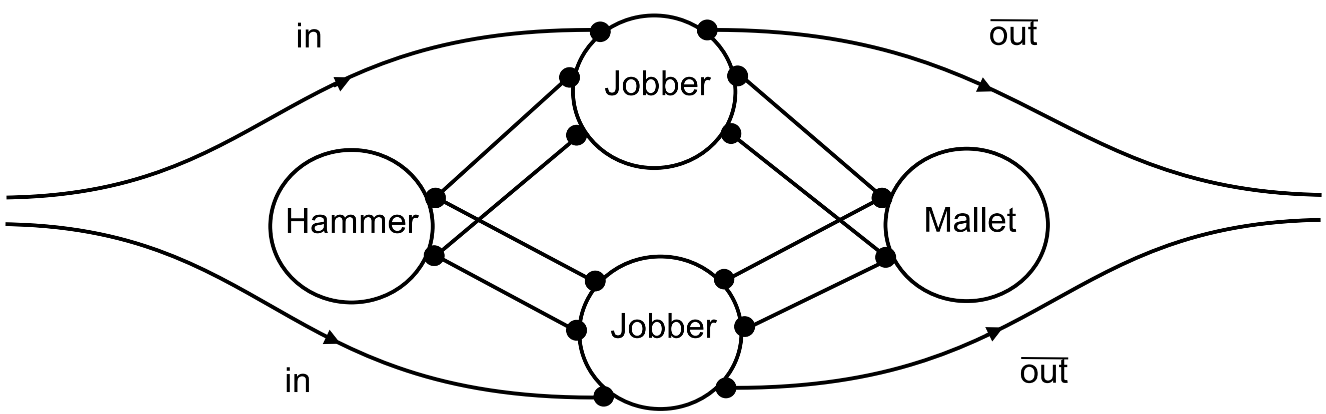
C = in(x).C'(x)

C'(x) = 'out(x).C

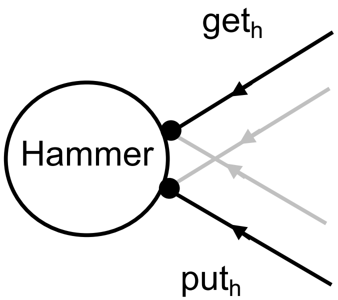
CCS Value-passing

Esempio: **Jobshop**



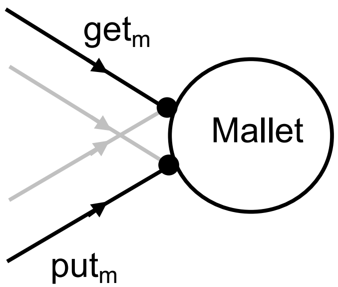


- un port è legato a più di un canale => i 2 jobber possono competere per l'uso di mazza e martello
- i 2 jobber competono per il prossimo job in arrivo sul canale in



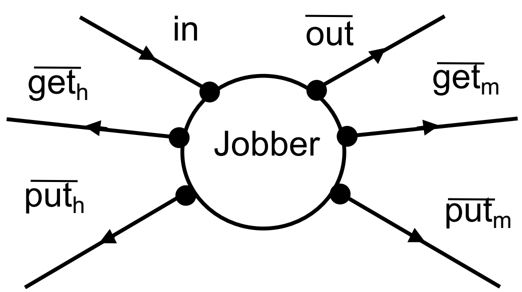
Hammer = $get_h.put_h$.Hammer

$get_h \simeq$ “accetto di essere preso”
put_h è il rilascio



Mallet = $get_m.put_m$.Mallet

Sto usando una versione di CCS “value-passing”, con if-then-else, derivabile da CCS “puro”



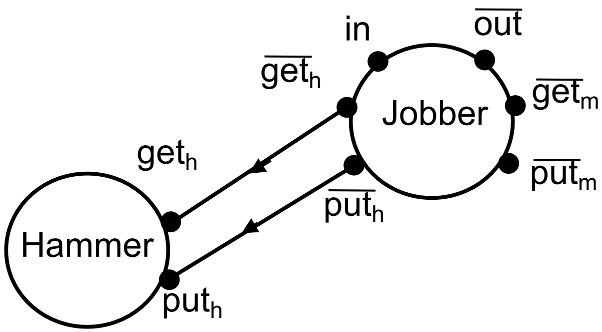
Jobber	=	in(job).Start(job)
Start(job)	=	if easy(job) then Finish(job) else
		if hard(job) then Usehammer(job)
		else Usetool(job)
Usetool(job)	=	Usehammer(job) + Usemallet(job)
Usehammer(job)	=	‘get_h.’put_h.Finish(job)
Usemallet(job)	=	‘get_m.’put_m.Finish(job)
Finish(job)	=	‘out(done(job)).Jobber

I job sono di tre tipi:
- easy (no tool)
- hard (hammer)
- neither (con tool)

object ricavato dal job

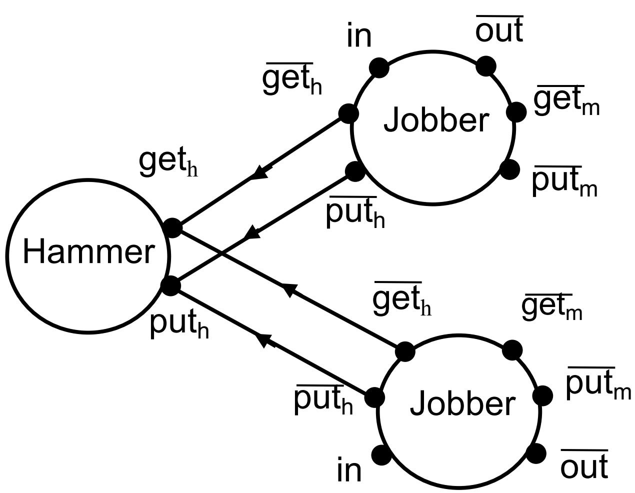
scelta guidata da chi è disponibile tra hammer e mallet

Jobber | Hammer



- " | " lega insieme i porti complementari, creando dei canali di comunicazione
- " | " è commutativa (come flow-graph!)

(Jobber | Hammer) | Jobber



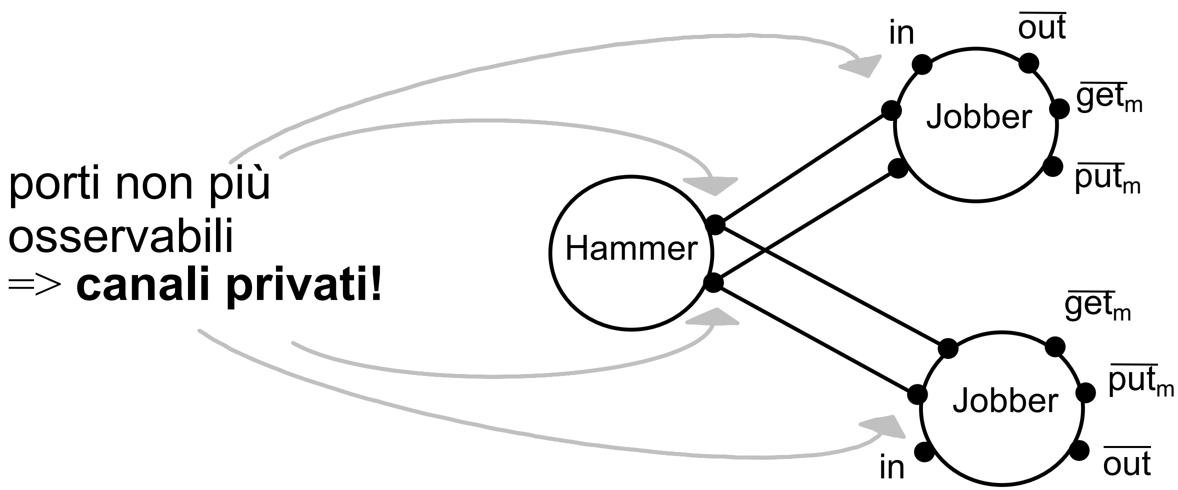
- " | " è associativa
- Jobber | (Hammer | Jobber) genera lo stesso flow-graph
- (Jobber | Jobber) | Hammer …..

Commutatività e associatività di " | " valgono per la rappresentazione a flow-graph; vedremo che varranno anche per (la maggior parte delle) equivalenze comportamentali

Come impedire che altri jobber possano utilizzare Hammer?
Rendendo non disponibili all'esterno le azioni di get_h e put_h!

OPERATORE DI “SCOPING” / RESTRIZIONE

$(\nu\ get_h)(\nu\ put_h)(Jobber \mid Hammer \mid Jobber)$

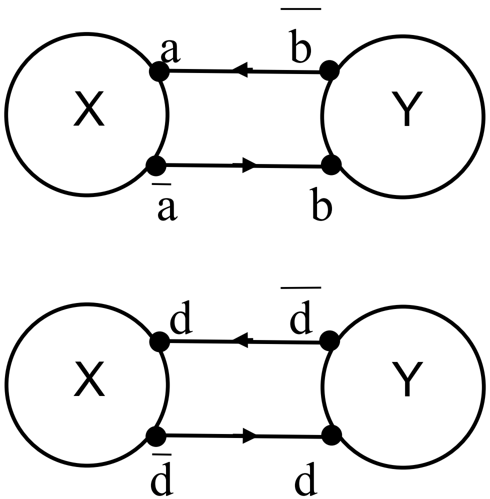


Jobshop = (ν L)(Jobber | Jobber | Hammer | Mallet)
dove $L = \{get_h, put_h, get_m, put_m\}$

Osservazione: vedremo che **Jobshop** (implementazione) è weak bisimile alla specifica **StrongJobber** = in(job).’out(done(job)).**StrongJobber**

Operatore Derivato:

Operatore di linking / pipeline



d è un nuovo nome!

$X \frown Y$ messa in parallelo con connessione "privata" su nomi non complementari.

In CCS si crea un canale solo tra nomi complementari => per X e Y è necessaria la ridenominazione

$$X \frown Y = (\nu d)(\, X\{^{d}/_{a}\} \mid Y\{^{d}/_{b}\}\,)$$

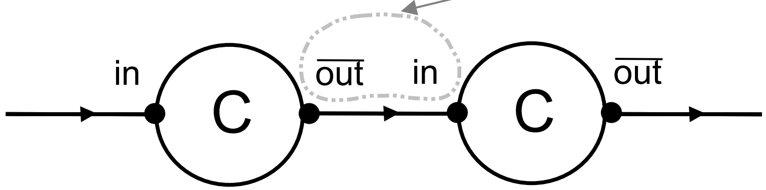
Osserva l'uso della sostituzione sintattica $\{^{d}/_{a}\}$ e $\{^{d}/_{b}\}$

Esempio d’uso del linking/pipeline: **buffer a due posizioni**

A = in(x).in(y).’out(x).’out(y).A

- Prima riempie il buffer e poi lo svuota, rispettando l'ordine
- A è un processo sequenziale
- **Versione parallela usando 2 buffer ad una posizione**

questa comunicazione non è visibile esternamente



$C \frown C = (vd)(\, C\,\{^{d}/_{out}\} \mid C\,\{^{d}/_{in}\}\,)$

- dopo la in sul primo C, si può verificare un out sul secondo C prima della successiva in => rispetta solo l'ordine

•Ricordate: C = in(x).’out(x).C

Sintassi formale (1): le azioni e le costanti

- Sia $\mathcal{L}$ un insieme contabile di azioni a, b, c, …(azioni di input)
- Sia $\mathcal{L}'$ l'insieme delle co-azioni 'a, 'b, 'c, …(azioni di output)
- $\mathcal{L} \cup \mathcal{L}'$ è l'insieme delle azioni osservabili α, β, γ, …. Con 'α intendiamo la co-azione di α, assumendo che ''$\alpha = \alpha$.
- Act = $\mathcal{L} \cup \mathcal{L}' \cup \{\tau\}$ insieme delle azioni μ, dove τ è l'azione non osservabile (o interna)
- $\mathcal{Cons}$ è un insieme contabile di costanti di processo (disgiunto da Act), A, B, C ….

Sintassi formale (2): termini del CCS

- Useremo 2 categorie sintattiche, P per processi sequenziali e Q per processi generali

  $P ::= 0 \ | \ \mu.Q \ | \ P + P$ processi sequenziali

  $Q ::= P \ | \ Q|Q \ | \ (\nu a)Q \ | \ C$ processi generali

  dove μ in Act, a in $\mathcal{L}$, e si assume che ogni costante C abbia definizione nella categoria sintattica Q, cioè C = q con q in Q.
- Assumiamo somma guardata (entrambi gli addendi sono sequenziali), quindi non è legale e.g. a.0 + (b.0 | c.0).
  - Il motivo è che è difficile concepire/implementare la scelta distribuita, inoltre
  - nessuna perdita in espressività *(tutti gli LTS finiti e tutte le reti di Petri finite si possono rappresentare solo con scelta guardata)*
- anche se da un punto di vista puramente formale/tecnico uno potrebbe usare una sola categoria sintattica:

  $Q ::= 0 \ | \ \mu.Q \ | \ Q+Q \ | \ Q|Q \ | \ (\nu a)Q \ | \ C$

Esempio di "non" termini

a.(A+B)   (b.0 + c.0).a.0   (va)(b.0) + c.0

(v τ)(a.τ.0)   (A|c.0) + a.B   0.a.0

(a.A + 'a.0).B   a.A.B

- Esempi di termini legali:

(va)(a.0)   (A|(vc)B)   a.(va)(a.0 | b.0)

Funzione Const(p): costanti che p può usare

Vogliamo calcolare l'insieme Const(p) delle costanti che un processo p può usare

Definition 3.1. (*Const*(p)**: Set of constants used by** p**)** With $Const(p)$ we denote the *least* set of process constants such that the following equations are satisfied:

$$Const(\mathbf{0}) = \emptyset \qquad Const(p_1 + p_2) = Const(p_1) \cup Const(p_2)$$

$$Const(\mu.p) = Const(p) \qquad Const(p_1 \mid p_2) = Const(p_1) \cup Const(p_2)$$

$$Const((\nu a)p) = Const(p) \qquad Const(A) = \begin{cases} \{A\} & \text{if } A \text{ undef.} \\ \{A\} \cup Const(q) & \text{if } A \stackrel{def}{=} q \end{cases}$$

A term p such that $Const(p)$ is finite is called *finitary*, as it uses finitely many process constants only. □

Example 3.2. Let us consider the simple vending machine specified as follows:

$$C_1 \stackrel{def}{=} coin.C_2 \quad C_2 \stackrel{def}{=} \overline{coffee}.C_1$$

It is not difficult to see that

$$\begin{aligned} Const(C_1) &= \{C_1\} \cup Const(coin.C_2) \\ &= \{C_1\} \cup Const(C_2) \\ &= \{C_1, C_2\} \cup Const(\overline{coffee}.C_1) \\ &= \{C_1, C_2\} \cup Const(C_1) \end{aligned}$$

so that the *least* set $Const(C_1)$ satisfying this recursive equation is $\{C_1, C_2\}$.

As a further example, consider process A_0, with the family of process constants $A_i \stackrel{def}{=} a_i.A_{i+1}$ for $i \in \mathbb{N}$; it is easy to observe that

$$\begin{aligned} Const(A_0) &= \{A_0\} \cup Const(a_0.A_1) &&= \{A_0\} \cup Const(A_1) \\ &= \{A_0, A_1\} \cup Const(a_1.A_2) &&= \{A_0, A_1\} \cup Const(A_2) \\ &= \{A_0, A_1, A_2\} \cup Const(a_2.A_3) &&= \{A_0, A_1, A_2\} \cup Const(A_3) \\ &= \dots \end{aligned}$$

so that the limit of this increasing sequence of finite sets is $Const(A_0) = \{A_i \mid i \in \mathbb{N}\}$, which is an infinite set. $\square$

Funzione Const(p) per processi finitari

The computation of $Const(p)$ may be much more intricate than in the examples above; a formal treatment is outside the scope of this introductory text. However, observe that, when p is finitary, there is an obvious algorithm to compute $Const(p)$: it is enough to remember all the constants that have been already met while scanning p, in order to avoid applying again function *Const* over their bodies. This can be achieved by the following auxiliary function δ that has, as additional parameter, a set I of already known constants:

$$
\begin{array}{rlrl}
\delta(\mathbf{0},I) & = \emptyset & \delta(p_1+p_2,I) & = \delta(p_1,I)\cup\delta(p_2,I) \\
\delta(\mu.p,I) & = \delta(p,I) & \delta(p_1\,|\,p_2,I) & = \delta(p_1,I)\cup\delta(p_2,I) \\
\delta((\nu a)p,I) & = \delta(p,I) & \delta(A,I) & = \begin{cases} \emptyset & A\in I, \\ \{A\} & A\notin I\wedge A \text{ undef.} \\ \{A\}\cup\delta(p,I\cup\{A\}) & A\notin I\wedge A\stackrel{def}{=}p \end{cases}
\end{array}
$$

Then, for any finitary term p, $Const(p)=\delta(p,\emptyset)$, where the additional parameter is the empty set because so is the set of process constants we assume to know at the beginning of the scanning of p.

Termini definiti

- Una costante A è detta definita se possiede una equazione di definizione A = p
- Un termine CCS q è detto fully defined se tutte le sue costanti in Const(q) sono definite.

La richiesta che q sia fully defined è ovviamente dettata dall'impossibilità altrimenti di associare una semantica al termine. Ad esempio, a.A può fare a e raggiunge lo stato A, ma se A non è definita …

Costanti guardate

- Una occorrenza di una costante A in un termine p è detta strongly guarded se tale occorrenza occorre in un sottotermine μ.q di p.

  Ad es. l'unica occorrenza di B in a.0 | a.B è strongly guarded mentre l'occorrenza di sinistra di A in A | a.A non lo è.

- Una costante (definita) A, con equazione A = p, è guardata se (1) ogni occorrenza di A in p è strongly guarded e (2) ogni altra costante B che occorre in p o è strongly guarded in p oppure è guardata.

  Ad es. B = b.B è guardata, ed anche A = (a.A|B) |c.C è guardata, mentre C = a.0 |C non lo è e neppure D = F e F = D (loop nella dimostrazione).

Perché costanti guardate?

- La richiesta di guardatezza non è indispensabile per lo sviluppo tecnico che segue, ma ci sono buone ragioni per richiederla:
  - Un termine guardato genera un lts finitely branching; infatti, ad esempio, $C = a.0|C$ genera addirittura un lts non image-finite!
  - Garantisce unicità di soluzione, modulo bisimulation equivalence ~, di equazioni di processi del tipo

    $$X \sim E(X)$$

    dove E(_) è un contesto. Ad es., $X \sim \tau.X + a.0$, dove X occorre guardata dal tau, ha una unica soluzione $A = \tau.A + a.0$

Sintassi formale (3): Processi CCS

- L'insieme $\mathcal{P}$ dei processi CCS è dato da tutti quei termini CCS p tali che ogni costante A in Const(p) è (definita e) guardata.
- (Nota: una costante guardata è pure definita).
- Un processo CCS p è finitario se Const(p) è finito. (Finitary CCS)

Calcoli finiti vs calcoli finitari

Poichè un termine p di CCS finitario usa solo un numero finito di costanti (ed anche solo un numero finito di azioni), il processo p può essere visto come un **sistema formale finito**. Così come un DFA che usa un insieme finito di stati e di simboli dell'alfabeto è un sistema formale finito.

Tuttavia, il linguaggio **finitary CCS** (ovvero l'insieme di tutti i processi di Finitary CCS), che usa nel suo complesso un insieme infinito di costanti e di azioni, **non è un sistema formale finito**; così come l'insieme di tutti i possibili DFA non è un sistema formale finito perche' usa infiniti stati ed infiniti simboli di alfabeto.

Più correttamente, uno dovrebbe definire la sintassi di CCS in modo parametrizzato rispetto ad un insieme finito di costanti C e ad un insieme finito di azioni L:

$$CCS_{C,L}$$

in modo che il sistema formale (ovvero il linguaggio) sia davvero definito in modo finito**.** **Ogni processo di $CCS_{C,L}$ è un sistema formale finito ed anche l'insieme (infinito) di tutti i processi di $CCS_{C,L}$ costituisce un sistema formale finito.**

Calcoli finiti vs calcoli finitari (2)

Quindi, un **calcolo finito** è un linguaggio parametrizzato rispetto ad un insieme finito di costanti C e ad un insieme finito di azioni L; ad esempio,

$$\mathbf{CCS_{C,L}}$$

Uno potrebbe evitare la parametrizzazione assumendo che C e L siano molto grandi, ma non infiniti. In questo modo un processo usa solo un numero finito di costanti e di azioni per costruzione.

Quando non è importante il nome effettivo delle azioni e delle costanti, indicheremo con **CCS(h,k)** il linguaggio che usa h costanti e k azioni al massimo.

Convenzioni di notazione (1)

- Sintassi "astratta" ambigua: $a.b.0 + c.0$ può rappresentare

  $(a.(b.0)) + (c.0)$ oppure $a.((b.0) + (c.0))$
- O si usano parentesi per disambiguare o, meglio, si assume che gli operatori abbiano una diversa priorità di parsing, secondo questo ordine:

  $(\nu a)\_ \;>\; \mu.\_ \;>\; \_|\_ \;>\; \_+\_$
- Perciò $a.b.0 + c.0$ denota il termine $(a.(b.0)) + (c.0)$;
- $(\nu\, a)b.c.0 \mid a.0$ denota il termine $((\nu a)(b.(c.0)))|(a.0)$.
- Ed anche $b.(\nu a)c.0 + a.b.0$ rappresenta il termine

  $(b.((\nu\, a)(c.0))) + (a.(b.0))$.

Convenzioni di notazione (2)

- Vedremo che sia la composizione parallela che la scelta sono associative (rispetto a quasi tutte le equivalenze comportamentali); perciò talvolta useremo i corrispondenti operatori n-ary al posto di quelli binari:

  $p_1 + p_2 + \dots + p_n$ abbreviato come $\Sigma_{1\leq k\leq n} p_k$

  $p_1 \mid p_2 \mid \dots \mid p_n$ abbreviato come $\Pi_{1\leq k\leq n} p_k$
- Per la restrizione,

  $(\nu a_1)(\nu a_2)\dots(\nu a_n)p$ abbreviato in $(\nu a_1 a_2 \dots a_n)p$ o anche in $(\nu L)p$, dove $L = \{a_1, a_2 \dots a_n\}$.
- Spesso ometteremo l'operatore **0** alla fine, sicchè, per es., $a.0|b.0$ viene abbreviato come $a|b$.

Convenzioni di notazione (3)

- Spesso omettiamo addendi inutili del tipo **0**, perché dimostreremo che **0** è l'elemento neutro rispetto alla somma per quasi tutte le equivalenze che abbiamo studiato. Ad esempio: (**0** + p) + **0** è abbreviato a p
- Usando l'operatore n-ario di somma e l'assunzione che gli addendi **0** vengano assorbiti, possiamo descrivere la sintassi di CCS in modo alternativo come segue:

$$p ::= \Sigma_{j \in J} \mu_j . p_j \mid p|p \mid (\nu a)p \mid C$$

with the assumption that J is finite and $\Sigma_{j \in J} \mu_j . p_j = \mathbf{0}$ when $J = \emptyset$. The sequential process terms are those of the form $\Sigma_{j \in J} \mu_j . p_j$.