Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali
Corso di Laurea Triennale in Informatica

Implementazione di Semantiche Operazionali per MULTI-CCS

Modelli e Sistemi Concorrenti

Relatore:
Chiar.mo Prof.
Roberto Gorrieri

Presentata da:
Giovanni Moschese

Correlatore:
Chiar.mo Dott.
Cristian Versari

Sessione III
Anno Accademico 2010/2011

Introduzione

La teoria della concorrenza è un insieme di concetti e tecniche, di carattere logico-matematico, finalizzate a descrivere e studiare il comportamento di sistemi formati da più processi (intuitivamente un processo è un insieme di azioni svolte in un certo ordine) che agiscono contemporaneamente e che interagiscono tra di loro.

Nella teoria della concorrenza, l'oggetto di studio è il processo, visto non come singola entità, ma come parte di un sistema attraverso il quale può interagire con altri processi. Utilizzando le parole di Edsger Dijkstra (1930 - 2002)

> "Concurrency occurs when two or more execution flows are able to run simultaneously"[1]

La concorrenza, intesa come contemporaneità di esecuzione di parti diverse di uno stesso programma, rappresenta una caratteristica di primaria importanza nello sviluppo dei sistemi reattivi -cioè quei sistemi che hanno come caratteristica quella di dover reagire a eventi provenienti dall'esterno- siano essi software di sistema e in particolare del Sistema Operativo, sia nel caso del software applicativo soppratutto con riferimento a specifici ambiti come, per esempio, quello relativo al software real-time o al software di rete. In tutti questi casi la concorrenza fornisce un ausilio fondamentale per raggiungere gli obiettivi che il sistema software si prefigge sollevando, per contro, una serie di problematiche e di caratteristiche non presenti nella

[1] La concorrenza accade quando due o più flussi di esecuzione sono in grado di procedere simultaneamente

classica programmazione sequenziale. Problemi come il deadlock, il livelock, la starvation, l'inconsistenza dovuta all'accesso a risorse condivise, la mutua esclusione, sono molto studiati in quanto di difficile individuazione. Quindi, dire che un sistema concorrente è corretto (ovvero scevro dai problemi sopra citati) è difficile da mostrare. Caratteristica invece auspicabile nei sistemi reattivi e assolutamente da evitare nella programmazione sequenziale è, ad esempio, la non terminazione.

Ecco dunque che si rende necessario studiare il problema con un maggior livello di astrazione, creando un modello matematico del sistema che si vuole implementare, e sul quale verranno dimostrate le proprietà di correttezza desiderate. Se il modello è abbastanza accurato, tali proprietà forniranno una solida garanzia sulla bontà del sistema in esame.

In letteratura sono stati utilizzati vari formalismi per modellare sistemi reattivi concorrenti, come riportato in [WN95]. Un di questi sono le algebre di processo.

Il termine algebre di processo è stato introdotto nel 1982 da *Bergstra* e *Klop*[BK84, BK85][2] per indicare un'algebra universale che soddisfa un particolare insieme di assiomi. Un'algebra di processo è un approccio algebrico che consente di studiare processi concorrenti e i cui strumenti sono linguaggi algebrici volti a descrivere processi.

L'obiettivo di tali algebre è catturare la nozione di comunicazione e di concorrenza, entrambe essenziali per la comprensione di sistemi dinamici complessi.

Una parte essenziale della teoria dei sistemi complessi è una nozione precisa e utilizzabile di comportamento. Il comportamento è il modo in cui un agente interagisce con il resto del sistema ed è ragionevole definirlo come le capacità di comunicazione del sistema stesso. In altre parole, il comportamento di un sistema è esattamente ciò che è osservabile, e osservare un sistema equivale a comunicare con esso. Le algebre di processo si focalizzano

[2] In bibliografia sono riportati solo due dei tanti articoli degli autori citati che hanno portato alla formalizzazione delle algebre di processo.

sulle transizioni osservabili.

Nelle algebre di processo un agente è identificato con le azioni che descrivono il suo comportamento. Ogni sua azione o è un'interazione con gli altri agenti del sistema, allora è una comunicazione, o agisce indipendentemente da loro, allora è in concorrenza. Solitamente ci sono due tipi di azioni: azioni di input e azioni di output. La comunicazione è vista come l'occorrenza simultanea di due azioni: una di input e l'altra di output. In un sottoprocesso un canale può essere utilizzato sia per comunicare con un altro sottoprocesso, sia con l'ambiente. Per descrivere un processo in cui i canali sono utilizzati solo per comunicazioni interne, si introduce la nozione di restrizione. La restrizione è molto potente e consente di astrarsi dai dettagli di funzionamento interno dei processi.

Un'attenzione particolare del formalismo, basato sulle algebre di processo, è rivolta agli operatori che costruiscono processi complessi derivandoli da quelli più semplici. Solitamente, nelle agebre di processo c'è un operatore binario di "parallelo", scritto "|", tale che se P e Q sono processi allora $P \mid Q$ è un processo che stabilisce il comportamento di P e di Q eseguiti parallelamente. Abbiamo quindi che una piccola collezione di operatori delle algebre di processi, permette di costruire gerarchicamente processi di complessità arbitraria. Questo metodo conduce ad un formalismo specifico che è di base, ma molto potente.

Una delle prime algebre di processo definite in letteratura è il CCS [Mil80] [Mil89] acronimo di *Calculus of Communicating Systems* a cura di Robin Milner. Il CCS è stato il linguaggio di riferimento di questa dissertazione. Da qui partiremo analizzando i suoi limiti, vale a dire analizzando quei problemi non affrontabili in questo formalismo e cercando quindi di sopperire a queste mancanze con la definizione di una sua estensione che chiameremo Multi-CCS, caratterizzata dall'introduzione di un ulteriore operatore unitario, rispetto agli operatori già presenti nel CCS.

Il Multi-CCS quindi sarà l'oggetto della tesi e sarà presentato in due varianti. La prima è la presentazione di una nuova semantica operazionale in-

terleaving con congruenza strutturale, daremo ampio risalto alle sue caratteristiche; la seconda versione è il Multi-CCS linear step, una versione più "lasca" per quanto riguarda l'equivalenza tra processi e meno "pesante" dal punto di vista computazionale, cioè una versione del linguaggio di modellazione con una semantica operazionale senza la congruenza strutturale ma con altre caratteristiche, tra cui quella di rendere la bisimulazione una congruenza rispetto al parallelo.

Contributo di questa tesi è l'implementazione di due tool che interpretano tali semantiche e la dimostrazione di equivalenza tra le semantiche dei due tool rispetto alla definizione dei linguaggi interpretati; mostrando come la presenza della congruenza strutturale tra le regole di semantica operazionale renda i due interpreti radicalmente diversi non solo da un punto di vista implementativo ma incida, chiaramente, anche sulla difficolta dimostrativa. Infatti la dimostrazione sull'equivalenza della semantica del Multi-muCCS (nome dell'interprete del Multi-CCS) e il suo linguaggio interpretato non è formalmente completa, ma viene data solo una dimostrazione informale.

Capitolo 1 - Il CCS In questo capitolo viene introdotto in maniera molto veloce uno dei linguaggi di modellazione concorrente più importanti, il CCS. Viene introdotta la sintassi, la semantica con le relative regole di semantica operazionale.

Capitolo 2 - Il Multi-CCS Il capitolo si propone di presentare il nuovo linguaggio e lo fa esponendo i motivi per cui si ha la necessità di ampliare il CCS. Quindi vengono evidenziati i limiti del CCS e il modo in cui il nuovo linguaggio di modellazione cerca di sopperire a questi limiti. Di conseguenza la presentazione formale della sintassi, della semantica e delle regole di semantica operazionale del Multi-CCS.

Capitolo 3 - Semantica del Multi-muCCS L'obiettivo di questo capitolo è il più importante di tutta la dissertazione. Si vuole in questa sezione dimostrare che è possibile implementare un interprete per il Multi-CCS.

Capitolo 4 - Interprete per Multi-CCS: Multi-muCCS Si analizza in questa sezione il modo in cui è stato implementato Multi-muCCS. Si analizza quindi implementazione dell'analizzatore sintattico e delle regole di semantica operazionale introdotte nel capitolo precedente e dimostrate essere equivalenti alle regole di semantica operazionale originali del Multi-CCS.

Capitolo 5 - Multi-CCS linear step: Multi-CCS-ls In questo capitolo si introduce la versione linear step del Multi-CCS

Capitolo 6 - Interprete per Multi-CCS-ls: Multi-muCCS-ls Si fornisce l'interprete per la semantica linear step per il Multi-CCS

Indice

Capitolo 1

CCS

In questo capitolo introduciamo il CCS[1] per descrivere i sistemi reattivi. Descriveremo la sua sintassi e la sua semantica (in termini di LTS) passando chiaramente per le sue regole di semantica operazionale (SOS). Il CCS è stato sviluppato da Robin Milner, presentato in [Mil80] e rivisto in [Mil89].

In generale per definire un linguaggio occorre specificare:

1. Dominio Sintattico
2. Dominio Semantico
3. Funzione Semantica

graficamente quindi si ha una schematizzazione come in fugura 1.1:

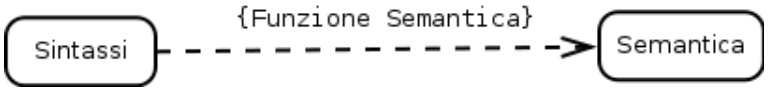

Figura 1.1: Schema di definizione di un linguaggio.

Introduciamo velocemente in questa sezione il CCS parlando quindi nell'ordine di:

1. LTS (Labelled Transition System), semantica;

[1] Calculus of Communicating Systems

2. CCS, sintassi;

3. SOS, collegamento tra sintassi e semantica.

1.1 Semantica LTS

Il dominio semantico di CCS è visto come un "automa". Nella teoria della concorrenza, più che di automi, si parla di *sistemi di transizione etichettati* (LTS dall'inglese Labeled Transition System). Le differenze principali tra i due formalismi sono:

1. negli automi il numero di stati è solitamente finito, mentre negli LTS no;

2. negli automi si usa la nozione di stato finale per descrivere il linguaggio accettato dall'automa, mentre negli LTS (poiché il linguaggio non è molto rilevante) non serve avere tale nozione

Def 1 (LTS)**.** *Un label transition system è una tripla* $TS = (Q, A, \rightarrow)$*, dove:*

- Q *è l'insieme degli stati*
- A *è l'insieme delle etichette (usualmente chiamate azioni)*
- $\rightarrow \subset Q \times A \times Q$ *è la una relazione di transizione*

Data una transizione $(q, a, q') \in \rightarrow$*,* q *è chiamato stato sorgente,* q′ *stato obiettivo e* a *etichetta della transizione.*

1.2 Sintassi, CCS

1.2.1 Descrizione informale

Diamo una informale visione degli operatori del CCS:

Nil: Il processo più semplice che si può definire in CCS è 0 (chiamato nil), il processo vuoto.

Azione prefissa: Se prefissiamo al processo 0 un'azione a, otteniamo $a.0$, che è il processo che può eseguire a per poi fermarsi. In generale le azioni prefisse operano facendo un'azione μ (che può essere un input a, un output $\overline{a}$, o un'azione invisibile τ) e un processo p per costruire il processo $\mu.p$. Questa è la più semplice forma di azioni sequenzializzate disponibile in CCS.

Con solo questi due operatori è possibile costruire solo processi terminanti.

Costante di processo: Se vogliamo dare un nome ad un processo p si può usare una costante di processo (usualmente denominata da una lettera maiuscola $A, B, C \ldots$). Per esempio, $B \stackrel{def}{=} a.b.0$ denota il processo finito che può eseguire la sequenza ab per poi fermarsi, così come $C \stackrel{def}{=} 0$, che denota il processo vuoto. Le costanti di processo sono molto importanti perché forniscono gli strumenti per esprimere la forma di comportamento ciclico quando la costante A occorre all'interno delle definizione del termine p. Ad esempio $A \stackrel{def}{=} b.A$.

Scelta: Con il processo vuoto, le azioni prefisse e le costanti si possono costruire LTS con al massimo una transione d'uscita per ogni stato. La possibilità di scegliere tra alcune azioni alternative è data dall'operatore $+$. Per esempio, con $a.b.0 + b.a.0$ descriviamo che il sistema può eseguire o la seguenza ab oppure la seguenza ba e poi fermarsi. In generale $p_1 + p_2$ è un processo che può eseguire o un'azione di p_1 per poi continuare con l'esecuzione di p_1 oppure eseguire un'azione di p_2 per poi continuare ad eseguire p_2.

Composizione parallela: Due processi indipendenti p_1 e sp_2 possono essere eseguiti in parallelo con l'aggiunta dell'operatore "|". Questo significa che i due processi possono avanzare nell'esecuzione in modo asincrono o interagire eseguendo azioni complementari di input/output in modo sincrono (chiamata "handshake synchronization"). Per esempio,

il processo $a.0 \,|\, \overline{a}.0$ può avanzare o in maniera asincrona, quindi effettuando le azioni a e $\overline{a}$ separatamente, oppure sincronizzarsi e produrre l'azione invisibile τ. La sincronizzazione è un'operazione strettamente binaria in CCS.

Restrizione: La restrizione dichiara nomi di canali privati. Il processo ristretto $(\nu\, a)p$ dichiara che l'azione con nome a non può essere usata per l'interazione con qualsiasi processo in parallelo con p e che a può solo essere usato per una sincronizzazione interna.

1.2.2 Sintassi formale

Def 2. *sintassi CCS:*

- $A = \{a, b, c, d \ldots\}$: *insieme infinito di azioni di (input);*
- $\overline{A} = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, \ldots\}$: *insieme infinito di azioni complementari (output);*
- $\mathcal{L} = A \cup \overline{A}$: *insieme di tutte le azioni visibili, denotati con* $\alpha, \beta \ldots$*;*
- $Act = \mathcal{L} \cup \{\tau\}$: *insieme delle azioni osservabili e non, con* τ *azione interna non osservabile, denotati con* μ*;*
- $K = \{A, B, C, D, \ldots, Z\}$: *insieme dei nomi di agenti (costanti di processo);*
- $p ::= 0 \,\big|\, \mu.q \,\big|\, p + p$: *processi sequenziali*
- $q ::= p \,\big|\, q \,|\, q \,\big|\, (\nu\, a)q \,\big|\, C$: *processi*

Nota 1. *Abbiamo purtroppo una discrepanza notazionale: la restrizione è rappresentata nel formalismo della sintassi in* $(\nu\, etichette)(processo)$*; negli* lts*, invece, la restrizione è rappresentata con il "\" (backslash):* $(processo)\backslash\{etichette\}$

1.3 Funzione Semantica SOS

La Semantica Operazionale Strutturata SOS lega CCS con LTS, quindi rende ogni processo definibile in CCS rappresentabile anche in LTS. Definiamo quindi il labeled transition system per il linguaggio CCS. A questo scopo ricorriamo alla tecnica di Plotkin chiamata SOS[2] [Plo04a, Plo04b] secondo la quale le transizioni sono definite tramite un sistema di inferenza composta di assiomi e regole, la cui definizione è syntax-driven[3].

Una tipica regola operazionale SOS ha la forma: $(\textsf{Regola})\ \dfrac{\text{premesse}}{\text{conclusione}}$, dove "premesse" è la congiunzione di zero (in tale caso la regola si chiama assioma) o più transizioni, mentre "conclusione" è una transizione[4].

Def 3 ($\mathsf{LTS}_{\mathsf{CCS}}$)**.** *Il label transition system per CCS, $\mathcal{C}$ è una tripla $(\mathcal{P}, Act, \rightarrow)$ dove:*

$\mathcal{P}$ è l'insieme di tutti i processi possibili in CCS espressi dalla sintassi CCS;

$\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ è la più piccola relazione di transizione generata dagli assiomi e regole della Tabella 1.1.

(Pref): È un assioma; dichiara che per ogni azione μ e per ogni processo p lo stato $\mu.p$ può eseguire una transizione etichettata μ raggiungendo p. Nota che l'evento del prefisso scompare nel raggiungere lo stato p (viene "consumato"); per questa ragione questa operazione viene chiamata dinamica.

(Cons): È una regola che stabilisce che una costante C può fare ciò che è predefinito dal corpo della sua definizione equazionale: se $C \stackrel{def}{=} p$ e $p \xrightarrow{\mu} p'$ allora $C \xrightarrow{\mu} p'$.

[2] Structural Operational Semantics.

[3] Guidata dalla sintassi.

[4] In logica matematica, una regola operazionale prende il nome di inferenza, in altre parole è uno strumento che permette di passare da un numero finito di proposizioni assunte come premesse ad una proposizione che funge da conclusione.

(Pref)	$\mu.p \xrightarrow{\mu} p$	(Cons)	$\dfrac{p \xrightarrow{\mu} p'}{C \xrightarrow{\mu} p'}$	$C \stackrel{def}{=} p$
(Sum_1)	$\dfrac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'}$	(Sum_2)	$\dfrac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$	
(Par_1)	$\dfrac{p \xrightarrow{\mu} p'}{p \,\vert\, q \xrightarrow{\mu} p' \,\vert\, q}$	(Par_2)	$\dfrac{q \xrightarrow{\mu} q'}{p \,\vert\, q \xrightarrow{\mu} p \,\vert\, q'}$	
(Com)	$\dfrac{q \xrightarrow{\overline{\alpha}} q' \qquad p \xrightarrow{\alpha} p'}{p \,\vert\, q \xrightarrow{\tau} p' \,\vert\, q'}$	(Res)	$\dfrac{p \xrightarrow{\mu} p'}{(\nu\, a)p \xrightarrow{\mu} (\nu\, a)p'}$	$\mu \neq a, \overline{a}$

Tabella 1.1: SOS: regole di inferenza per il CCS.

(Sum_1): Chiarisce perché diciamo che le regole sono definite per induzione sulla struttura del termine; questa regola stabilisce che, al fine di ricavare una transizione dallo stato $p + q$, noi risolviamo prima il problema più semplice di cercare una transizione da p: se $p \xrightarrow{\mu} p'$ allora $p + q \xrightarrow{\mu} p'$.

(Sum_2): È simmetrica alla Sum_1.

Nota: l'operatore di scelta scompare una volta effettuata la scelta (viene "consumata"), per questo motivo anche la scelta è detta dinamica.

(Par_1): La regola descrive l'esecuzione asincrona di una azione da parte di uno dei due sottocomponenti del processo parallelo: se $p \xrightarrow{\mu} p'$ allora $p \,|\, q \xrightarrow{\mu} p' \,|\, q$. Si noti che il sottocomponente q non è scartato dalla transizione e per questa ragione l'operatore di *Par* viene chiamato statico.

(Par_2): È simmetrico a Par_1.

(Com): Questa regola descrive come può avvenire l'interazione tra i due sottocomponenti della composizione parallela. Se i sottotermini possono eseguire azioni input/output complementari allora la sincronizzazione è possibile e la transizione risultante non può essere usata per inter-

azioni future, essendo etichettata da τ. Vedremo nel capitolo 2, come l'esecuzione multiway (una sincronizzazione tra più di due processi) arricchisce il CCS.

(Res): Spiega che il ruolo di una restrizione è di legare un nome così che non è liberamente disponibile per l'ambiente esterno. Chiaramente $(\nu\, a)p$ impedisce ogni transizione con a e $\overline{a}$ che p potrebbe produrre, mentre non ha effetto sulle altre transizioni di p. Anche la restrizione è un operatore statico.

Esempio 1 (coffe machine)**.**
Definiamo i due agenti, il computer scientist (CS) e la coffee macheine (CM):

$CS \stackrel{def}{=} \overline{pub}.CS1$

$CS1 \stackrel{def}{=} \overline{coin}.CS2$

$CS2 \stackrel{def}{=} caffe.CS$

$CM \stackrel{def}{=} coin.CM1$

$CM1 \stackrel{def}{=} \overline{caffe}.CM$

Il processo CS prima pubblica, poi, per poter pubblicare ancora è costretto a consumare una moneta e a prende un caffè. La coffee machine invece deve ricevere una moneta per porer rilasciare un caffè. Gli lts *dei due processi distinti sono riportati in Figura 1.2.*

*L'*lts *del parallelo $CM \mid CS$ é riportato in Figura 1.3.*

*Applicando la restrizione sui canali di coin e coffee abbiamo invece l'*lts *riportato in figura 1.4*

Milner, quindi con il suo CCS, assegna a ogni costrutto sintattico del calcolo una semantica operazionale, introducendo le nozioni di derivazione e di albero di derivazione. Definisce una nozione di equivalenza tra agenti basata intuitivamente sull'idea che due agenti sono distinguibili se un osservatore

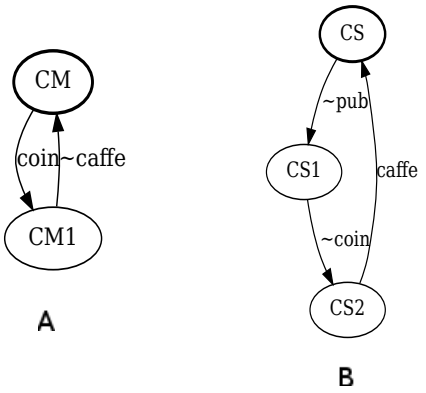

Figura 1.2: (A) lts CM (B) lts CS

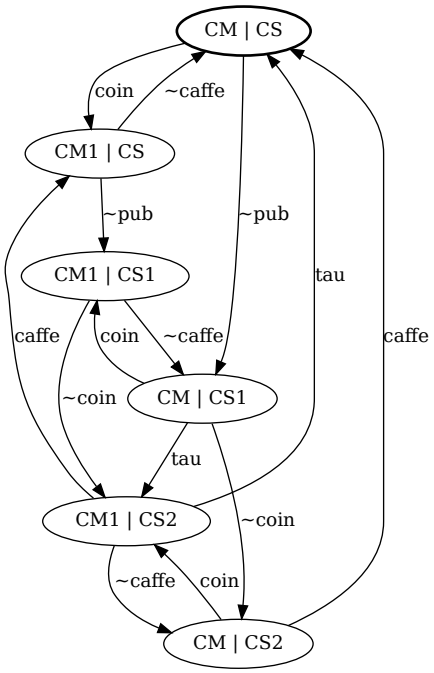

Figura 1.3: lts del processo $CM \,|\, CS$

esterno che può solo interagire con loro, è in grado di distinguerli. A partire da questa idea, introduce le relazioni di strong equivalence e di observation equivalence. In particolare mostra che la prima è una relazione di congruenza, cioè che è preservata da tutti i contesti algebrici.

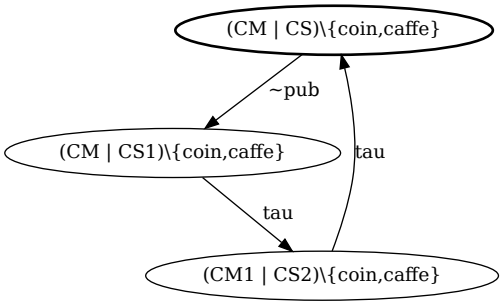

Figura 1.4: lts del processo $(\nu\, coin, caffe)CM \,|\, CS$

Equivaleze comportamentali[AILS07]

È possibile descrivere un sistema concorrente, scritto in uno dei linguaggi come il CCS, ad un livello più alto di astraione. L'idea alla base è quella di stabilire quando due programmi, scritti in CCS, si comportano allo stesso modo da un punto di vista dell'osservatore esterno. Accenniamo quindi quelle che sono le possibili equilavenze comportamentali dando solo le loro definizioni.

Def 4 (Isomorfismo ($\cong$)). *Siano* $TS_1 = (Q_1, A_1, \rightarrow_1, q_1)$ *e* $TS_2 = (Q_2, A_2, \rightarrow_2, q_2)$ *due radici di sistemi di transizione. Un isomorfismo è una funzione* $f : Q_1 \rightarrow Q_2$ *tale che:*

- $q \xrightarrow{a} q'$ *iff* $f(q) \xrightarrow{a} f(q')$
- $q_2 = f(q_1)$

Def 5 (Traces). *Sia* $TS = (Q, A, \rightarrow, q_0)$ *una radice di un sistema di transizione. Una traccia di* TS *è una (possibilmente vuota) sequenza di azioni* $a_1 \ldots a_n$ *tale che* $q_0 \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q_n$. *In altre parole, l'insieme delle traccie di* TS *è* $Tr(TS) = \{\sigma^1, \ldots, \sigma^n \in A | \exists q' \in Q . q_0 \xrightarrow{\sigma^1, \ldots, \sigma^n} q\}$.
$TS_1 = TS_2$ *se* $Tr(TS_1) = Tr(TS_2)$

Def 6 (Simulation). *Una* simulazione tra TS_1 e TS_2 è una relazione $R \subseteq (Q_1 \times Q_2)$ tale che se $(q_1, q_2) \in R$ allora $\forall a \in (A_1 \cup A_2)$

- $\forall q_1'$ tale che $q_1 \xrightarrow{a} q_1'$, $\exists q_2'$ tale che $q_2 \xrightarrow{a} q_2'$ e $(q_1', q_2') \in R$;

Se $\mathsf{TS}_1 = \mathsf{TS}_2$ noi sappiamo che R è una simulazione su TS_1. Uno stato q è simulato da q', denotato $q \overset{<}{\sim} q'$ se esiste una simulazione R tale che $(q, q') \in R$. Due stati q e q' sono una simulation equivalent, denotato con $q \simeq q$ se $q \overset{<}{\sim} q'$ e $q' \overset{<}{\sim} q$.

Def 7 (bisimulazione)**.** *Una* bisimulazione tra TS_1 e TS_2 è una relazione $R \subseteq (Q_1 \times Q_2)$ tale che se $(q_1, q_2) \in R$ allora $\forall a \in (A_1 \cup A_2)$

- $\forall q_1'$ tale che $q_1 \xrightarrow{a} q_1'$, $\exists q_2'$ tale che $q_2 \xrightarrow{a} q_2'$ e $(q_1', q_2') \in R$;
- $\forall q_2'$ tale che $q_2 \xrightarrow{a} q_2'$, $\exists q_1'$ tale che $q_1 \xrightarrow{a} q_1'$ e $(q_1', q_2') \in R$;

Se $TS_1 = TS_2$ sapiamo che R è una bisimulazione su TS_1 Due stati q e q' sono bisimili, denotato $q \sim q'$, se esiste una bisimulazione R tale che $(q, q') \in R$

Capitolo 2

Multi-CCS

In questa sezione presentiamo il Multi-CCS, un'estensione del CCS ottenuta dall'introduzione di un altro operatore di prefixing[1] $\underline{\mu}.p$ (chiamato strong prefixing in opposizione al normale prefixing $\mu.p$), con la capacità di esprimere transizioni altrimenti inesprimibili in CCS, come la sincronizzazione multi-party.

Vista l'importanza e la definizione non standard dell'argomento, consigliamo la lettura dell'appendice A di questo documento, dove verrà trattata l'α-conversione. Presentiamo quindi alcuni casi di studio su come Multi-CCS affronta problemi classici nella teoria della concorrenza, come i "filosofi a cena" e il problema dei "lettori scrittori".

2.1 Neo del CCS

Il CCS, presentato nel Capitolo 1, è Turing-completo (cioè possiede l'abilità di calcolare tutte le funzioni calcolabili). Questo non significa che il linguaggio di modellazione è in grado di risolvere qualsiasi tipo di problema concorrente. La Turing-completezza, uper-bound per la programmazione sequenziale, non è abbastanza per garantire la risolubilità di tutti i prob-

[1] Un' altra estensione del CCS da cui deriva l'idea del Multi-CCS è il A^2CCS [GMM90, GM90].

lemi della teoria concorrente. Vedremo che, per esempio, una soluzione in Multi-CCS del famoso problema dei filosofi a cena [Dij71] (vedi qui di seguito per dettagli), assumendo l'acquisizione atomica dei due bastoncini, non può essere eseguito in CCS.

Questa ossevazione solleva un'importante questione: qual è il formalismo *completo* per la concorrenza? E rispetto a cosa? Sfortunatamente non siamo ancora in grado di dare una risposta a queste filosofiche questioni.

2.2 I filosofi a cena

Il famoso problema, proposto da Dijkstra in [Dij71], è definito come segue: Cinque filosofi siedono attorno ad una tavola rotonda, con un piatto privato e cinque bastoncini; ogni bastoncino è condiviso dai due filosofi vicini. I filosofi possono pensare e mangiare: per mangiare un filosofo deve acquisire entrambi i bastoncini che egli condivide con i suoi vicini, partendo dal bastoncino alla sua sinistra e quindi quello alla sua destra. Poichè tutti i filosofi si comportano allo stesso modo, il problema è intrinsecamente simmetrico.

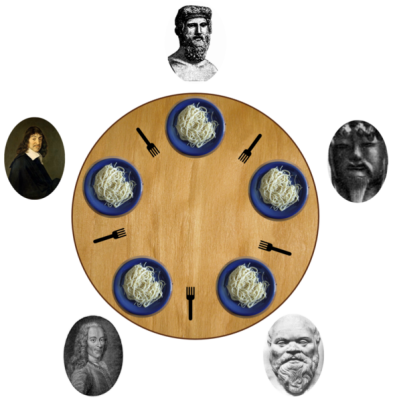

Figura 2.1: Descrizione visiva dei filosofi a cena.

Un tentativo di soluzione in CCS di questo problema può essere il seguente, dove per semplicità consideriamo il sottoproblema con solo due filosofi e due bastoncini.

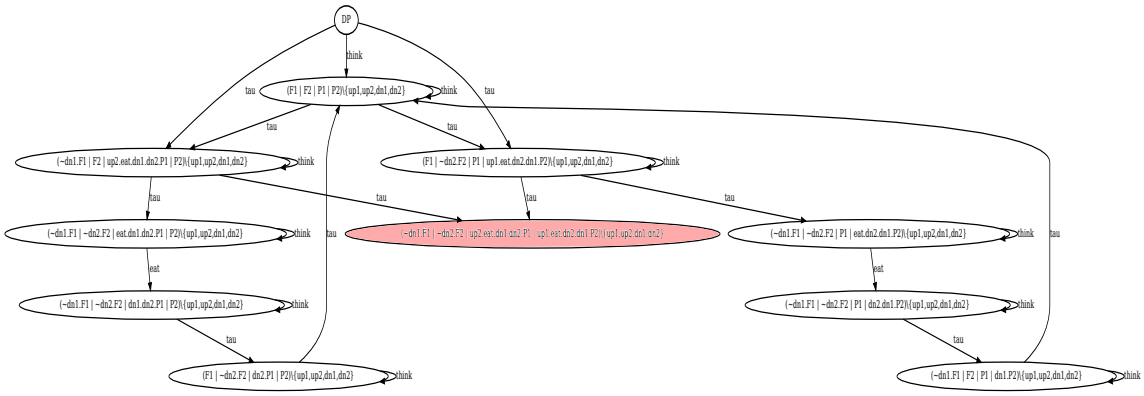

Figura 2.2: LTS dei filosofi a cena in CCS, soluzione simmetrica.

I bastoncini possono essere definiti con la costante $fork_i$:

$$fork_i \stackrel{def}{=} \overline{up_i}.\overline{dn_i}.fork_i \qquad \text{per } i = \{0, 1\}$$

I due filosofi possono essere descritti con $phil_i$:

$$phil_i \stackrel{def}{=} think.phil_i \; + \; up_i.up_{i+1}.eat.dn_1.dn_{i+1}.phil_i \qquad \text{per } i = \{0, 1\}$$

dove $i + 1$ è da calcolare modulo 2. Il sistema totale è DP definito come:

$$DP \stackrel{def}{=} (\nu\, L)(((phil_0 \,|\, phil_1) \,|\, fork_0) \,|\, fork_1)$$
$$\text{dove } L = \{up_0, up_1, dn_0, dn_1\}$$

il cui LTS è definito in Figura 2.2.

Chiaramente questa soluzione "istintiva" causa deadlock esattamete quando i due filosofi prendono i bastoncini alla loro sinistra nello stesso istante di tempo, restando quindi in attesa del bastoncino alla loro destra, in Figura 2.2 questo stato viene indicato di rosso.

Una nota soluzione alternativa a questo problema è di spezzare la simmetria, invertendo l'ordine di acquisizione dei bastoncini per l'ultimo filosofo. Nel nostro ristretto caso con due filosofi e due bastoncini avremo che:

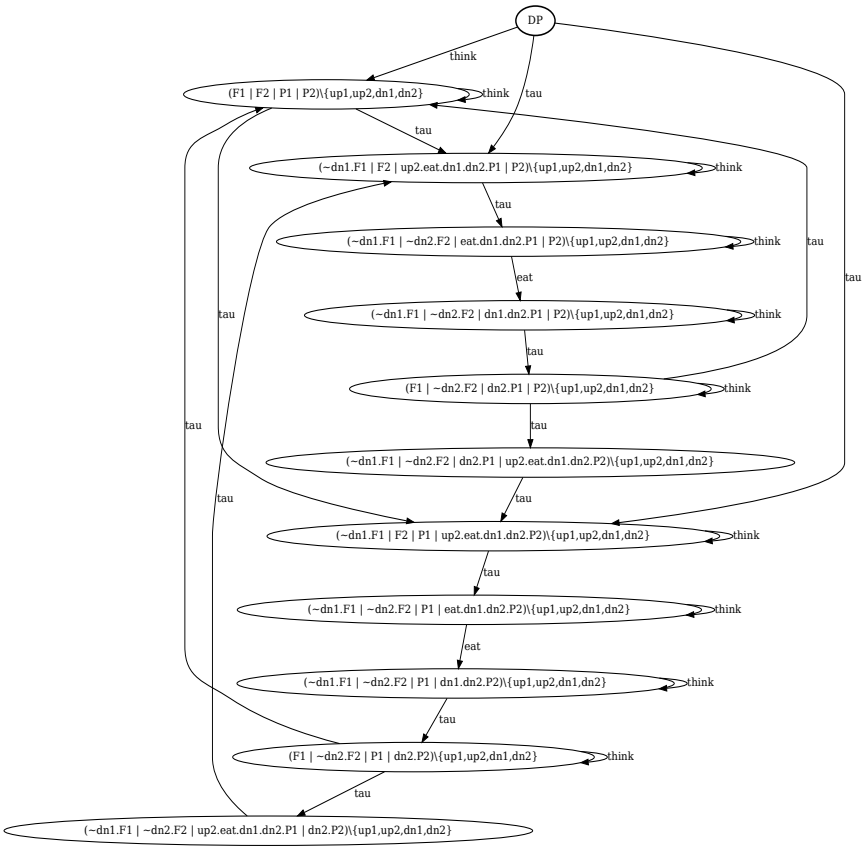

Figura 2.3: LTS dei filosofi a cena in CCS, soluzione non simmetrica.

$$phil'_0 \stackrel{def}{=} (think.phil'_0 \ + \ up_0.up_1.eat.dn_0.dn_1.phil_0)$$
$$phil'_1 \stackrel{def}{=} (think.phil'_1 \ + \ up_0.up_1.eat.dn_0.dn_1.phil_1)$$

il sistema totale quindi diventa:

$$DP' \stackrel{def}{=} (\nu\, L)(((phil'_0 \,|\, phil'_1) \,|\, fork_0) \,|\, fork_1)$$
$$\text{dove } L = \{up_0, up_1, dn_0, dn_1\}$$

il cui LTS è definito in Figura 2.3.

Questa soluzione funziona correttamente (non è introdotto deadlock) ma non è conforme alla specifica che richiede che tutti i filofosi siano definiti allo stesso modo.

Una famosa soluzione è quella di definire l'acquisizione dei bastoncini, da parte dei filosofi, in modo atomico. Quindi un filosofo o prende entranbi i bastoncini o nessuno.

Questo requisito può essere approssimato formalmente in CCS come segue:

$$phil''_i \stackrel{def}{=} (think.phil''_i + up_i.(dn_i.phil''_i + up_{i+1}.eat.dn_i.dn_{i+1}.phil''_i))$$
$$\text{per } i = \{0, 1\}$$

dove, nel caso in cui il secondo bastoncino non sia disponibile, il filosofo può mettere giù il primo bastoncino e ritornare allo stato iniziale. Tuttavia il nuovo sistema è il seguente:

$$DP'' \stackrel{def}{=} (\nu\, L)(((phil''_0 \,|\, phil''_1) \,|\, fork_0) \,|\, fork_1)$$
$$\text{dove } L = \{up_0, up_1, dn_0, dn_1\}$$

il cui LTS è raffigurato in Figura 2.4. In questa soluzione i filosofi sono liberi da deadlock ma possono ora divergere: i due filosofi possono essere impegnati in un interminabile livelock[2] perché la lunga operazione di acquisizione dei due bastincini può sempre fallire.

Sfortunatamente, una soluzione che implementi correttamente l'acquisizione atomica dei due bastoncini non può essere programmata in CCS perché è privo di ogni costrutto per l'atomicità che consentirebbe anche una sincronizzazione a più vie tra il filosofo e i due bastoncini. Infatti, Francez e Rodeh propongono in [FR80] una soluzione distribuita, simmetrica, deterministica per il problema dei filosofi a cena in CSP [Hoa85], sfruttando la sua capacità di sincronizzazione a più vie. Inoltre, Lehmann e Rabin dimostrano che tale soluzione non può esistere in un linguaggio con solo sincronizzazioni binarie come il CCS [LR81]. Quindi, se vogliamo risolvere il problema dei filosofi a cena in CCS, dobbiamo estendere la sua capacità in qualche modo.

[2]Una condizione che si verifica quando due o più processi cambiano continuamente il loro stato in risposta ai cambiamenti in altri processi. Il risultato è che nessuno dei processi verrà completato.

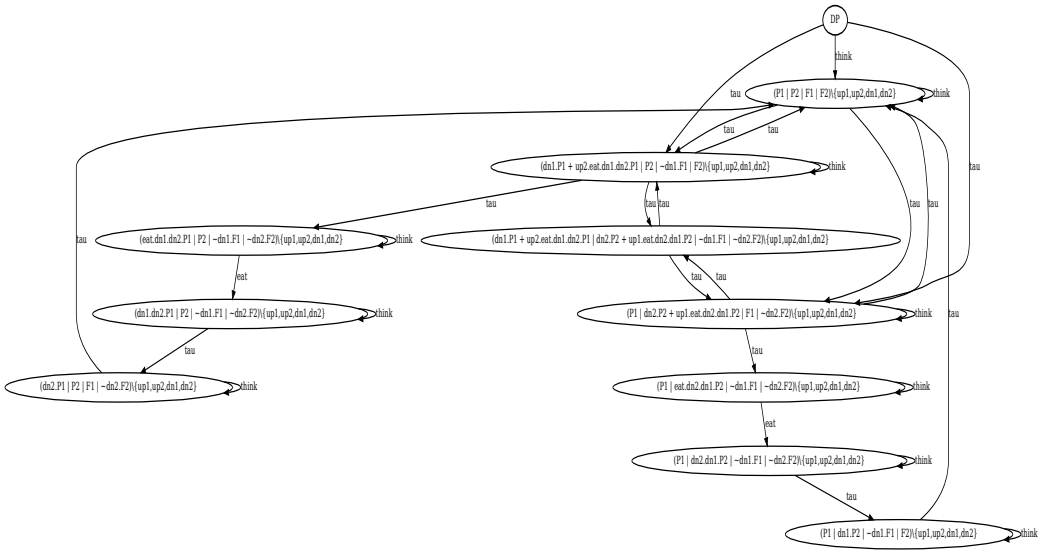

Figura 2.4: LTS dei filosofi a cena in CCS, soluzione con livelock.

2.3 Strong prefixing: un operatore per l'atomicità

Arricchiamo il CCS con l'aggiunta di un operatore $\underline{\mu}.p$, chiamato STRONG PREFIXING, dove μ è la prima azione di una transizione che continua con p (a condizione che p possa compiere la transizione). Le regole SOS per il nuovo operatore sono riportate in Tabella 2.1, dove σ è una non vuota sequenza di azioni. La regola $(S - Pref_3)$ permette la creazione di transizione etichettate da sequenze di azioni non vuote: per muovere $\underline{\mu}.p$ è necessario che p possa effettuare una transizione, cioè il resto della transazione. Quindi $\underline{\mu}.0$ non può effettuare alcun'azione. Usualmente se un'azione è etichettata da $\sigma = \alpha_1, \ldots, \alpha_{n-1}, \alpha_n$ allora tutte le azioni $\alpha_1, \ldots, \alpha_{n-1}$ sono dovute alla STRONG PREFIXING, mentre α_n all'operazione prefissa normale (o α_n è l'ultima azione STRONG PREFIXING prima di di una τ).

Le regole $(S-Pref_1)$ $(S-Pref_2)$ assicurano che mosse τ non vengano mai aggiunte in una sequenza σ, garantendo quindi che una transizione $p \xrightarrow{\sigma} p'$ o $\sigma = \tau$ o σ sia composta solo da azioni visibili, cioè σ che varia sull'insieme

$$(\mathsf{S}-\mathsf{Pref}_1)\ \frac{p \xrightarrow{\sigma} p'}{\underline{\tau}.p \xrightarrow{\sigma} p'} \quad (\mathsf{S}-\mathsf{Pref}_2)\ \frac{p \xrightarrow{\tau} p'}{\underline{\alpha}.p \xrightarrow{\alpha} p'} \quad (\mathsf{S}-\mathsf{Pref}_3)\ \frac{q \xrightarrow{\overline{\alpha}} q' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\alpha\sigma} p'}$$

Tabella 2.1: Regole SOS per l'operatore di STRONG PREFIXING in Multi-CCS

$\mathcal{A} = (\mathcal{L} \cup \mathcal{L})^+ \cup \{\tau\}$.

2.3.1 Utilizzo dell'operatore strong prefixing

Affrontiamo ora il problema dai filosofi in Multi-CCS implementando la soluzione "acquisizione atomica dei bastoncini". Grazie all'utilizzo del nuovo operatore STRONG PREFIXING, possiamo descrivere i due filosofi nel seguente modo[3]:

$$phil_i \stackrel{def}{=} (think.phil_i + \underline{up_i}.up_{i+1}.eat.\underline{dn_1}.dn_{i+1}.phil_i) \qquad \text{per } i = \{0, 1\}$$

dove $i+1$ è da intendere modulo 2 e la sequenza atomica $up_i up_{i+1}$ modella l'acquisizione atomica dei due bastoncini. Per semplicità assumiamo che anche il rilascio dei bastoncini sia atomico, anche se quasto non è necessario per la correttezza. L'LTS per $phil_i$ è mostrato in Figura 2.5.

Cosa succede quando mettiamo un processo $\underline{\mu}.p$ in parallelo con un altro processo? Per esempio, se prendiamo $q = \underline{a}.b.0 \,|\, c.0$ l'ovvia generalizzazione della regola (Par_1) e (Par_2) della Tabella 1.1 garantisce che l'LTS per q è riportato in Figura 2.6. Se confrontiamo questo LTS con quello riportato in Figura 2.7, per il processo $q' = a.b.0 \,|\, c.0$ notiamo che la sequenza ab non può essere "interleaved" con l'azione c dell'altro componente della composizione parallela, quindi l'atomicità è realmente garantita.

[3]La definizione di $fork$ resta quella definita in 2.2.

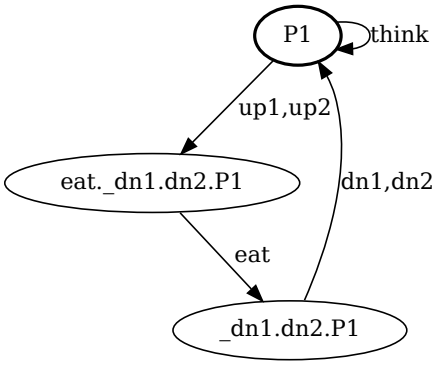

Figura 2.5: LTS del filosofo in Multi-CCS.

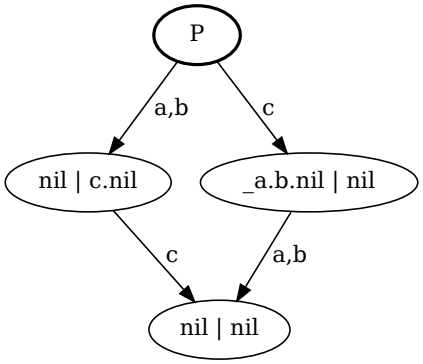

Figura 2.6: LTS del processo q.

2.4 Sincronizzazione in Multi-CCS

La regola (Com) della Tabella 2.1 deve essere estesa poiché ora le transizioni sono etichettate da sequenze di azioni. La nuova regola è:

$$(\mathsf{S}-\mathsf{Com})\ \frac{p \xrightarrow{\sigma_1} p' \qquad q \xrightarrow{\sigma_2} q'}{p \,|\, q \xrightarrow{\sigma} p' \,|\, q'} \quad Sync(\sigma_1, \sigma_2, \sigma)$$

come si può notare, la regola ha una condizione a margine sulla possibile sincronizzazione di σ_1 e σ_2 il cui risultato può essere σ.

Quando dovrebbe essere valido $Sync(\sigma_1, \sigma_2, \sigma)$? Poiché $(S - Com)$ è una generalizzazione di (Com) dovremmo richiedere che almeno una sincronizzazione abbia luogo. Quindi, assumendo che e.g. σ_2 sia composta da una

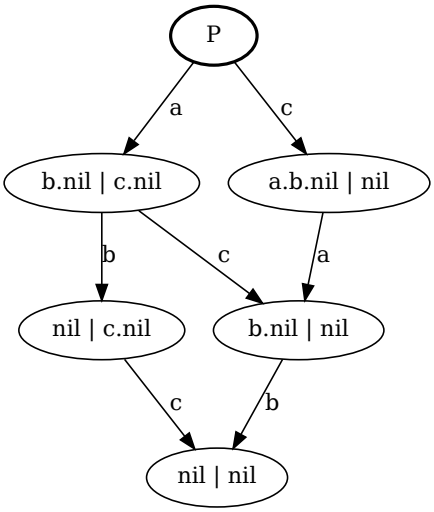

Figura 2.7: LTS del processo q'.

singola azione, diciamo α, questa deve sincronizzarsi con un'azione occorrente $\overline{\alpha}$ in $\sigma_1 = \sigma'\overline{\alpha}\sigma''$. Il risultato σ è proprio $\sigma'\sigma''$ se almeno una delle due azioni non è vuota, altrimenti (se $\sigma_1 = \overline{\alpha}$) il risultato è τ. Nota che quando il risultato σ non è τ allora esso può essere usato per un'ulteriore sincronizzazione con altre composizioni parallele, perettendo quindi la sincronizzazione multi-party.

2.4.1 Utilizzo dell'operatore di sincronizzazione

Prima introduciamo la definizione della regola SOS di interleaving *Sync* a margine della regola SOS (S-Com). Quindi $Sync(\sigma_1, \sigma_2, \sigma)$ risulta verificata se σ è ottenuta da un interleaving (con possibile sincronizzazione) di σ_1 e σ_2, dove almeno una sincronizzazione risulta avvenuta. La relazione *Sync* è definita dalle regole induttive in Tabella 2.3, che fanno uso della relazione ausiliaria *Int* che è come *Sync* senza la necessità che almeno una sincronizzazione abbia luogo.

Siscronizzazione multi-party

Mostriamo un utilizzo della sincronizzazione multi-party. Consideriamo tre processi:

- $p = \underline{a}.a.p'$
- $q = \overline{a}.q'$
- $r = \overline{a}.r'$

abbiamo quindi $P \stackrel{def}{=} ((p \,|\, q) \,|\, r)$ e $P' \stackrel{def}{=} ((p' \,|\, q') \,|\, r')$. Mostriamo in Figura l'albero di derivazione Π che dimostra la transizione $P \stackrel{\tau}{\to} P'$.

$$\dfrac{\dfrac{\dfrac{a.p' \stackrel{a}{\to} p'}{\underline{a}.a.p' \stackrel{aa}{\to} p'}\,(\text{s\_Pref}) \qquad \overline{a}.q' \stackrel{\overline{a}}{\to} q'}{\underline{a}.a.p' \,|\, \overline{a}.q' \stackrel{a}{\to} p' \,|\, q'}\,(Com)III \qquad \overline{a}.r' \stackrel{\overline{a}}{\to} r'}{((\underline{a}.a.p' \,|\, \overline{a}.q') \,|\, \overline{a}.r') \stackrel{\tau}{\to} ((p' \,|\, q') \,|\, r')}\,(Com)II$$

Le regole a margine delle deduzioni sono:

$$(Com)II = Sync(a, \overline{a}, \tau)$$

$$(Com)III = \frac{Int(a, \epsilon, a)}{Sync(a.a, \overline{a}, a)}$$

Completamento dei filosofi a cena con Multi-CCS

Dopo aver implementato la vita del filosofo, permettendo quindi l'acquisizione contemporanea dei bastoncini, definiamo il sistema completo dei filosofi DP come segue:

$$DP \stackrel{def}{=} (\nu\, L)(((phil_0 \,|\, phil_1) \,|\, fork_0) \,|\, fork_1) \quad \text{dove } L = \{up_0, up_1, dn_0, dn_1\}$$

La semantica operazionale genera un LTS a stati finiti per DP, rappresentato in Figura 2.8.

2.5 Sintassi e semantica operazionale

Come per il CCS, assumiamo di avere un insieme numerabile A di nomi di canali, il suo complementare $\overline{A}$, l'insieme $\mathcal{L} = A \cup \overline{A}$ di azioni visibili $(\alpha, \beta, \ldots)$ e l'insieme di tutte le azioni, $Act = \mathcal{L} \cup \{\tau\}$ tale che $\tau \notin \mathcal{L}$ (μ).

I termini di processo sono generati della grammatica definita in Def 8, dove utilizziamo due categorie sintattiche: p, per rappresentare processi sequenziali, e q per rappresentare ogni altro tipo di processo.

Def 8. *Termini di processi*

- $p ::= 0 \mid \mu.q \mid \underline{\mu}.q \mid p + p$: *processi sequenziali*
- $q ::= p \mid q \,|\, q \mid (\nu\, a)q \mid C$: *processi*

Dove l'unica differenza con la grammatica del CCS, Def 2, è l'operatore strong prefixing.

La semantica operazionale per il Multi-CCS è data dall'LTS definito come: $(\mathcal{P}, \mathcal{A}, \rightarrow)$, dove gli stati sono rappresentati da $\mathcal{P}$, $\mathcal{A} = (\mathcal{L}^+) \cup \{\tau\}$ è l'insieme delle etichette (rappresentate da σ), e $\rightarrow \subset \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ è la più piccola relazione di transizione generata dalle regole in Tabella 2.2.

Le nuove regole (S-Pref$_1$), (S-Pref$_2$), (S-Pref$_3$) e (S-Com) sono state già discusse. La regola (S-Res) è leggermente differente dal CCS, essa richiede che nessuna azione in σ possa essere a o $\overline{a}$. Con $n(\sigma)$ denotiamo l'insieme di tutte le azioni occorrenti in σ. Vi è un'ulteriore nuova regola chiamata (Cong), che fa uso di una congruenza strutturale $\equiv$, necessaria per superare una mancanza della composizione parallela: senza la regola (Cong) la composizione parallela non è associativa.

Spieghiamo questo concetto ritornando alla sincronizzazione multiparty presentato nel paragrafo 2.4.1. Riconsideriamo i tre processi p, q, r già definiti come: $p = \underline{a}.a.p'$, $q = \overline{a}.q'$, $r = \overline{a}.r'$. Abbiamo già dimostrato tramite l'albero di derivazione Π la sincronizzazione $P \xrightarrow{\tau} P'$. Tuttavia, se ora consideriamo il processo $P_1 \stackrel{def}{=} (p \,|\, (q \,|\, r))$, possimo vedere che non è possibile sincronizzare

(Pref)	$\mu.p \xrightarrow{\mu} p$	(S-Pref$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{\underline{\tau}.p \xrightarrow{\sigma} p'}$	
(S-Pref$_2$)	$\dfrac{p \xrightarrow{\tau} p'}{\underline{\alpha}.p \xrightarrow{\alpha} p'}$	(S-Pref$_3$)	$\dfrac{q \xrightarrow{\overline{\sigma}} q' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\alpha\sigma} p'}$	
(Sum$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{p + q \xrightarrow{\sigma} p'}$	(Sum$_2$)	$\dfrac{q \xrightarrow{\sigma} q'}{p + q \xrightarrow{\sigma} q'}$	
(Par$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{p \mid q \xrightarrow{\sigma} p' \mid q}$	(S-Com)	$\dfrac{q \xrightarrow{\sigma_2} q' \qquad p \xrightarrow{\sigma_1} p'}{p \mid q \xrightarrow{\sigma} p' \mid q'}$	$Sync(\sigma_1, \sigma_2, \sigma)$
(Par$_2$)	$\dfrac{q \xrightarrow{\sigma} q'}{p \mid q \xrightarrow{\sigma} p \mid q'}$	(S-Res)	$\dfrac{p \xrightarrow{\sigma} p'}{(\nu\, a)p \xrightarrow{\sigma} (\nu\, a)p'}$	$a, \overline{a} \notin n(\sigma)$
(Cong)	$\dfrac{p \equiv p' \xrightarrow{\sigma} q' \equiv q}{p \xrightarrow{\sigma} q}$			

Tabella 2.2: SOS: assiomi di inferenza per il Multi-CCS.

il processo p con i processi q e r. Questo è dimostrato dal seguente albero di derivazione.

$$\frac{\underline{a}.a.p' \xrightarrow{aa} p' \qquad \overline{a}.q' \mid \overline{a}.r' \xrightarrow{\overline{a}\overline{a}} q' \mid r'}{(\underline{a}.a.p' \mid (\overline{a}.q' \mid \overline{a}.r')) \xrightarrow{\tau} (p' \mid (q' \mid r'))}$$

Abbiamo quindi dimostrato che la multi-sincronizzazione dipende dalla associatività della composizione parallela, da qui la necessità di introdurre una congruenza strutturale insieme alla regola (Cong).

L'associatività è un'importante proprietà che ogni operatore di associazione parallela dovrebbe godere; noi supereremo questa lacuna introducendo una congruenza strutturale "$\equiv$" e una regola operazionale associativa (Cong). Dati un insieme di assiomi E, la congruenza strutturale $\equiv_E \subset \mathcal{P} \times \mathcal{P}$ è la congruenza indotta dagli assiomi in E. In altre parole $p \equiv_E q$ sse

$Sync(\alpha, \overline{\alpha}, \tau)$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma)}{Sync(\alpha\sigma_1, \overline{\alpha}\sigma_2, \sigma)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \tau)}{Sync(\alpha\sigma_1, \sigma_2, \alpha)}$
$\dfrac{Sync(\sigma_1, \sigma_2, \tau)}{Sync(\sigma_1, \alpha\sigma_2, \alpha)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Sync(\alpha\sigma_1, \sigma_2, \alpha\sigma)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Sync(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$
$Int(\alpha, \overline{\alpha}, \tau)$	$Int(\alpha, \epsilon, \alpha)$	$Int(\epsilon, \alpha, \alpha)$
$\dfrac{Int(\sigma_1, \sigma_2, \sigma)}{Int(\alpha\sigma_1, \overline{\alpha}\sigma_2, \sigma)}$	$\dfrac{Int(\sigma_1, \sigma_2, \tau)}{Int(\alpha\sigma_1, \sigma_2, \alpha)}$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Int(\alpha\sigma_1, \sigma_2, \alpha\sigma)}$
$\dfrac{Int(\sigma_1, \sigma_2, \tau)}{Int(\sigma_1, \alpha\sigma_2, \alpha)}$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Int(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$	

Tabella 2.3: Relazioni di sincronizzazione "$Sync$" e di Interleaving "Int".

$E \vdash p = q$, cioè p può essere dimostrato uguale a q tramite sistemi di equazioni deduttive $D(E)$.

La regola (Cong) fa uso di congruenza strutturale $\equiv$ su termini di processi indotti dalle quattro regole in Tabella 2.4, dove si assume che queste regole siano applicate solo a termini ground (i.e. la condizione a margine di ogni assioma deve essere soddisfatta dalle istanze ground degli assiomi).

E1	associatività	$(p \mid (q \mid r)) \equiv ((p \mid q) \mid r)$	
E2	commutatività	$p \mid q \equiv q \mid p$	
$E3_1$	Scope1	$(\nu\, a)(p \mid q) \equiv (p \mid (\nu\, a)q)$	se a non è libera in p
$E3_2$	Scope2	$(\nu\, a)(p \mid q) \equiv ((\nu\, a)p \mid q)$	se a non è libera in q
E4	α-conversione	$(\nu\, a)p \equiv (\nu\, b)p\{b/a\}$	se $b \in n(x)$
E5	costante	$C \equiv p$	se $C \stackrel{def}{=} p$

Tabella 2.4: Assiomi per la congruenza strutturale.

Il primo assioma E1 è per l'associatività dell'operatore parallelo; il secondo, E2, tratta la proprietà commutativa sempre dell'operatore parallelo; il terzo, $E3_1$ e $E3_2$, consistono dell'allargamento dello scope della restizione; il

quarto E4, è chiamato regola di α-conversione (vedi Appendice A); il quinto assioma invece spiega perché non abbiamo nessuna esplicita regola SOS per la gestione delle costanti.

La regola (Cong) allarga l'insieme delle transizioni derivabili da un dato stato p, infatti applicando questi due strumenti al processo P_1, abbiamo il seguente albero di derivazione.

$$\dfrac{(\equiv_5)\ \dfrac{(p \mid (q \mid r)) \equiv ((p \mid q) \mid r)}{P_1 \equiv P} \qquad \begin{array}{c}\Pi \\ P \xrightarrow{\tau} P'\end{array} \qquad \begin{array}{l}(\equiv_1)\ \dfrac{(p' \mid (q' \mid r')) \equiv ((p' \mid q') \mid r')}{\ldots} \\ (\equiv_5)\ \dfrac{((p' \mid q') \mid r') \equiv (p' \mid (q' \mid r'))}{P' \equiv P'_1}\end{array}}{(Cong)\ \dfrac{}{P_1 \xrightarrow{\tau} P'_1}}$$

Esempio 2. *Consideriamo $R \stackrel{def}{=} \underline{a}.c.A$, dove $A \stackrel{def}{=} \overline{a}.0 \mid \overline{c}.0$. Senza la regoal (Cong) e l'assioma E5 della Tabella 2.4, la derivazione $R \xrightarrow{\tau} 0 \mid 0 \mid 0$ non può essere possibile. L'lts è mostrato in figura 2.9.*

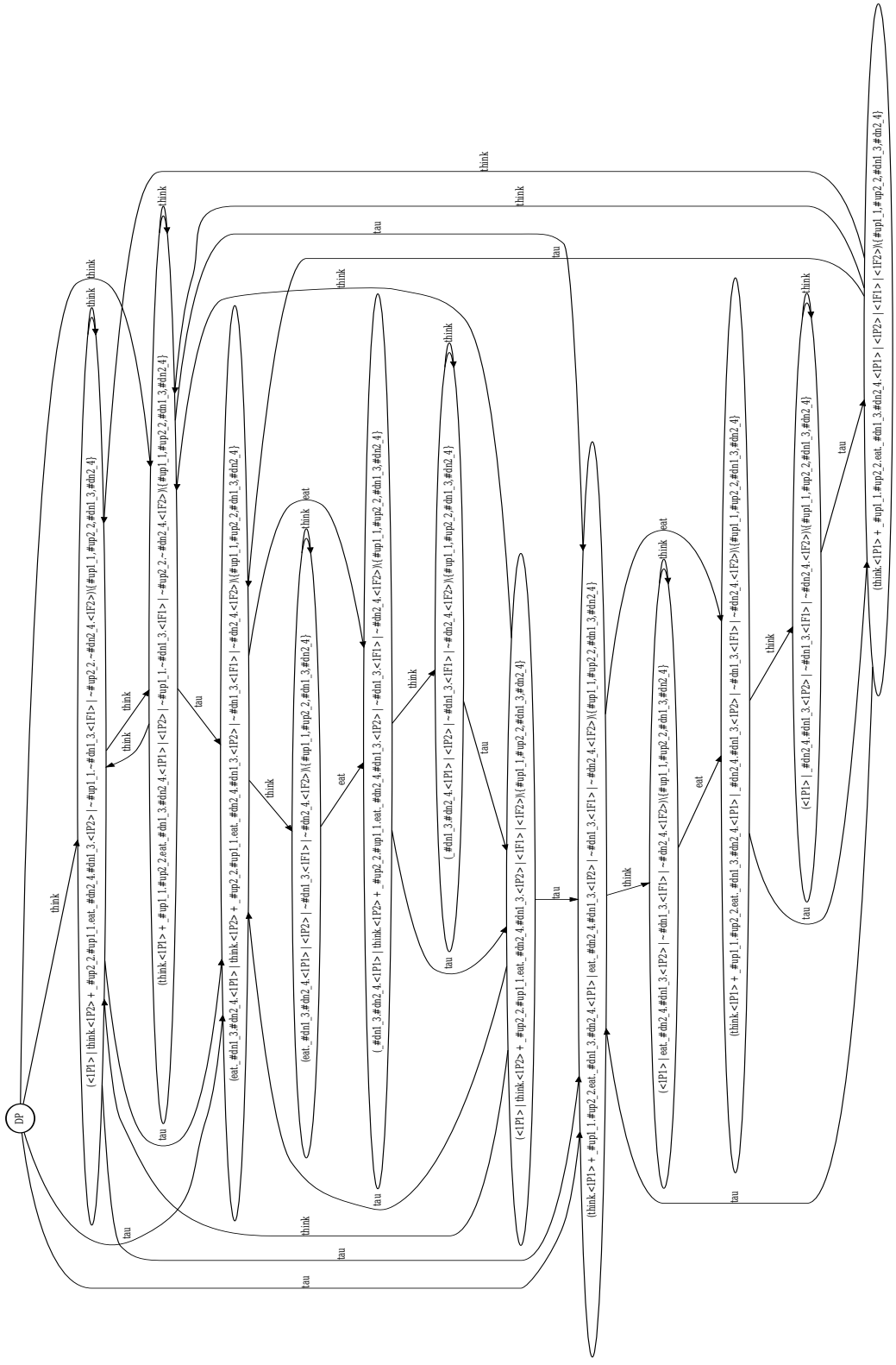

Figura 2.8: LTS dei filosofi a cena in Multi-CCS.

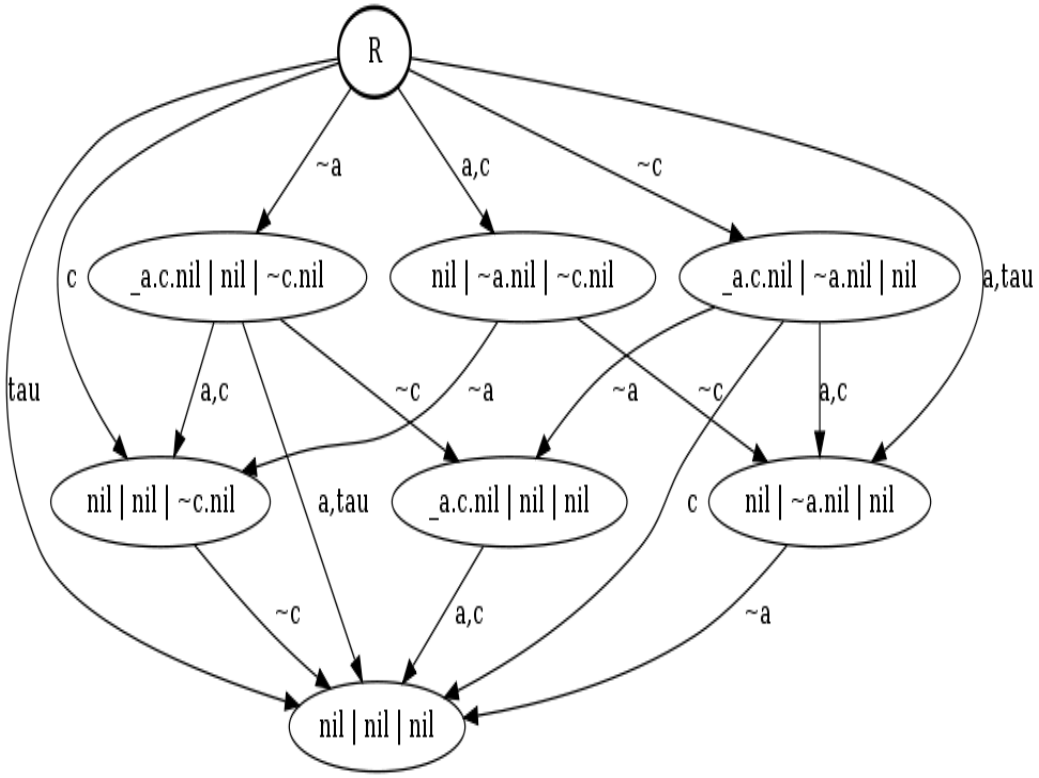

Figura 2.9: Lts dell'esempio 2.

Capitolo 3

Semantica del Multi-muCCS

Il seguente capitolo è il più importante di questa dissertazione, poiché pone le basi dimostrative del nostro lavoro, vale a dire l'implementazione di un interprete per il Multi-CCS; quindi dimostriamo che la semantica del Multi-muCCS (il nostro interprete Multi-CCS) equivale alla semantica formale del Multi-CCS data nel Capitolo 2.

3.1 Introduzione

3.1.1 Congruenza

Definiamo le proprietà caratteristiche del predicato di congruenza, queste possono essere descritte dal punto di vista logico nel seguente modo:

1. $\forall p \in \mathcal{P} \quad p \equiv p$;
2. $\forall p, q \in \mathcal{P} \quad p \equiv q \rightarrow q \equiv p$
3. $\forall p, q, r \in \mathcal{P} \quad p \equiv q \wedge q \equiv r \rightarrow p \equiv r$
4. $\forall p, q \in \mathcal{P} \quad p \equiv q \rightarrow \mathbb{C}[p] \equiv \mathbb{C}[q]$ vale a dire chiusura per contesti:
   (a) $\mu.p \equiv \mu.q \; \forall \mu \in Act$
   (b) $\underline{\mu}.p \equiv \underline{\mu}.q \; \forall \mu \in Act$

(c) $p + r \equiv q + r \ \forall r \in \mathcal{P}$

(d) $r + p \equiv r + q \ \forall r \in \mathcal{P}$

(e) $p \mid r \equiv q \mid r \ \forall r \in \mathcal{P}$

(f) $r \mid p \equiv r \mid q \ \forall r \in \mathcal{P}$

(g) $(\nu\, \mu)p \equiv (\nu\, \mu)q$

Le prime tre regole (la prima è un assioma) asseriscono che la relazione "$\equiv$" è una relazione di equivalenza, l'ultima regola che è una congruenza. Schematizzando si ha:

$$congruenza = \begin{cases} equivalenza = \begin{cases} preordine = \begin{cases} (1)\ Riflessivita' \\ (2)\ Transitivita' \end{cases} \\ (3)\ Simmetria \end{cases} \\ (4)\ Congruenza \end{cases}$$

Il Multi-CCS fa uso nella regola (*cong*) e anche degli assiomi di congruenza strutturali, introdotti nel Capitolo 2 e riproposti in Tabella 3.3. Quindi abbiamo la tabella 3.1

E1	associatività	$\mathbb{C}[((p \mid q) \mid r)] \equiv \mathbb{C}[(p \mid (q \mid r)))]$	
E2	commutatività	$p \mid q \equiv q \mid p$	
$E3_1$	Scope1	$\mathbb{C}[(\nu\, a)(p \mid q)] \equiv \mathbb{C}[((\nu\, a)p \mid q))]$	se a non è libera in p
$E3_2$	Scope2	$\mathbb{C}[(\nu\, a)(p \mid q)] \equiv \mathbb{C}[(p \mid (\nu\, a)q))]$	se a non è libera in q
E4	α-conversione	$\mathbb{C}[(\nu\, a)p] \equiv \mathbb{C}[(\nu\, b)p\{a/b\}]$	se $b \notin n(p)$
E5	costante	$C \equiv p$	se $C \stackrel{def}{=} p$

Tabella 3.1: Assiomi per la congruenza strutturale.

Def 9 (Classe CA (Commutativa-Associativa))**.** *Sia p un processo appartenente all'insieme dei "processi legali" in Multi-CCS ($P_{Multi-CCS}$), la sua*

classe CA (o classe di congruenza rispetto al primo e al secondo assioma di congruenza) $[p]_{\equiv_{1-2}}$ *è definita come:*

$$[p]_{\equiv_{1-2}} = \{x \in P_{Multi-CCS} | x \equiv_1 p \wedge x \equiv_2 p\}$$

cioè è l'insieme di tutti i processi x *che sono congruenti ad* p *secondo la prima e seconda equazione di congruenza strutturale per il Multi-CCS (Tabella 3.3).*

Nota 2. *: Se due classi hanno un elemento in comune, allora coincidono. Inoltre, ogni processo* p *sta in una ed una sola classe, da cui segue che se* $p \not\equiv_{1-2} q$*, allora le loro classi CA sono disgiunte.*

Def 10 (Operatore *pre*)**.** *Questo predicato implementa l'assioma per le costanti. Il predicato è molto semplice, in pratica sostituisce con la sua definizione la costante presente nel processo con un passo di riscrittura. Quindi:*

- $pre(p) = p'$
- $pre(0) = 0$
- $pre(\mu.p) = \mu.p$
- $pre(p + q) = pre(p) + pre(q)$
- $pre(p \mid q) = pre(p) \mid pre(q)$
- $pre((\nu\, \sigma)p) = (\nu\, \sigma)pre(p)$
- $pre(A) = pre(q)$ *se* $A \stackrel{def}{=} q$

3.1.2 Forma normale

Spesso è utile poter trasformare un processo legale in un'altro ad esso strutturalmente congruente che ha una qualche forma canonica prestabilita. Tipicamente ciò si realizza sostituendo una componente del processo dato, con altri processi ad esso congruenti, fino al raggiungimento della forma desiderata. Tale forma canonica è abitualmente detta *normale*, in quanto

il procedimento di riscrittura dei sottocomponenti non può essere applicato ulteriormente. La forma normale si definisce come una forma prenessa[1] associativa a sinistra rispetto all'operatore di composizione parallela. Quindi per ogni restrizione presente nell' espressione originale, a partire da quella più interna, si applicano la seconda e terza equazione di congruenza strutturale per il Multi-CCS allargando, quindi lo scope della restrizione senza modificare la semantica intesa, e poi, applicando la prima equazione di congruenza si trasforma il processo in forma associativa sinistra.

Def 11 (Forma normale)**.** *Un processo p appartenente ai processi legali del Multi-CCS è in forma normale se ha la forma* $(\nu x_1)(\nu x_2)(\nu x_3)\ldots(\nu x_n)pre(p_1)$ *con* $n \geqslant 0$*, dove* ν *è il nostro solito simbolo di restrizione e* p_1 *un processo che non contiene restrizioni ed è in forma associativa sinistra rispetto all'operatore di parallelismo. Quindi:*

$$p_1 = ((((p_1^1 \mid p_1^2) \mid p_1^3) \mid \ldots) \mid p_1^m)$$
$$(\nu\, x_1)(\nu\, x_2)(\nu\, x_3)\ldots(\nu\, x_n)\ \ pre(p_1)$$

Abbiamo che $(\nu\, x_1)(\nu\, x_2)(\nu\, x_3)\ldots(\nu\, x_n)$ *viene detto* prenex*, metre* p_1 matrice *della formula p.*

La forma normale di un processo viene costruita applicando ricorsivamente ad ogni restrizione presente in p la E_2 e E_3 equazione di congruenza strutturale nel seguente modo:

1. si cambia il nome di ogni canale legato ad ogni restrizione (a partire dalla restrizione più interna) con un nome di canale "fresco" (passo di α-conversione, regole di riscrittura definite in Appendice A);

2. si applica l'allargamento dello scope delle restrizioni presenti in p (sicuri a questo punto di non catturare canali in precedenza non legati), portando verso l'esterno tutte le restrizioni.

[1]Nella logica del primo ordine è utile considerare una forma normale detta prenessa, in cui tutti i quantificatori compaiono "in testa" alla formula. In questo ambito la forma prenessa viene intesa sostituendo al posto dei quantificatori classici le restrizioni, rispettando le regole di sostituzione sintattica elencate in Appendice A.

3. quindi riscrivendo la matrice del processo in forma associativa sinistra rispetto all'operatore di composizione parallela applicando la riscrittura multipasso dell'operatore $\rightharpoonup_{as}$, operatore definito nel seguente modo:

$$\mathbb{C}\{(P \mid (Q \mid R))\} \rightharpoonup_{as} \mathbb{C}\{((P \mid Q) \mid R)\}$$

la riscrittura multipasso ($\overset{*}{\rightharpoonup}_{as}$) invece è la chiusura transitiva e riflessiva di quella ad un passo, definita quindi nel seguente modo:

$$p \overset{*}{\rightharpoonup}_{as} p^n \quad \text{sse} \quad p \rightharpoonup_{as} \ldots \rightharpoonup_{as} p^n \quad \text{per qualche} \quad n \geq 0$$

Definiamo altri due operatori che ci serviranno per l'implementazione dell'interprete:

Def 12 (Operatore $\rightharpoonup_{com}$)**.** *Questo operatore permette di navigare tutta la classe commutativa del processo a cui è applicato. Matematicamente quindi l'operatore è definito come da secondo assioma di congruenza:*

$$\mathbb{C}\{p \mid q\} \rightharpoonup_{com} \mathbb{C}\{q \mid p\}$$

chiaramente definiamo la chiusura transitiva e riflessiva dell'operatore nel seguente modo:

$$p \overset{*}{\rightharpoonup}_{com} p^n \quad \textit{sse} \quad p \rightharpoonup_{com} \ldots \rightharpoonup_{com} p^n \quad \textit{per qualche} \quad n \geq 0$$

Def 13 (Operatore $\rightharpoonup_{ca}$)**.** *Per semplicità definiamo un unico operatore $\rightharpoonup_{ca}$ che chiamiamo "commutativa associativa sinistra", in grado di navigare le due classi definite precedentemente, quindi, altro non è che l'annidamento tra gli operatori $\rightharpoonup_{as}$ e $\rightharpoonup_{com}$*

Teorema 1. *I seguenti asserti sono veri:*

1. $((\nu\,\alpha)p' \mid p'') \equiv (\nu\,\alpha_{new})(p'\{\alpha_{new}/\alpha\} \mid p'')$
2. $((\nu\,\alpha^1)p' \mid (\nu\,\alpha^2)p'') \equiv (\nu\,\alpha^1_{new}, \alpha^2_{new})(p'\{\alpha^1_{new}/\alpha^1\} \mid p''\{\alpha^2_{new}/\alpha^2\})$

Nota 3. *σ_{new} è una variabile fresca non presente in p.*

Nota 4. *La secondo asserto è superfluo, perché è derivabile dal primo. Abbiamo deciso di inserirlo per completezza d'informazione.*

Dimostrazione. Segue dall'applicazione della sostituzione sintattica senza cattura e dalla degola di α-conversione (vedi Appendice A). □

Teorema 2 (Esistenza della forma normale)**.** *Per ogni processo p legale al Multi-CCS, esiste una forma normale $\{\!\{p\}\!\}$ tale che $p \equiv \{\!\{p\}\!\}$, con $\equiv$ congruenza strutturale per il Multi-CCS:*

$$\forall p \in \mathcal{P}_{Multi-CCS} \quad \exists \{\!\{p\}\!\} \; t.c. \quad p \equiv \{\!\{p\}\!\}$$

Dimostrazione. Per induzione sulla struttura di p.

caso base :

$p = nil$ è già in forma normale.

caso induttivo :

- $p = \alpha.p'$ con α un qualunque canale della forma $\alpha, \overline{\alpha}, \underline{\alpha}, \overline{\underline{\alpha}}$. Abbiamo come ipotesi induttiva, l'esistenza della forma normale di p rappresentata con $\{\!\{p'\}\!\}$, tale che $\{\!\{p'\}\!\} \equiv p'$; quindi per la definizione di forma normale e in base agli assiomi di congruenza strutturale per il Multi-CCS abbiamo che $\alpha.\{\!\{p'\}\!\}$ è già in forma normale.
- $p = p' + p''$. Abbiamo in questo caso come ipotesi induttiva l'esistenza delle due forme normali di p e q rappresentate con $\{\!\{p'\}\!\}$ e $\{\!\{p''\}\!\}$, tale che $\{\!\{p'\}\!\} \equiv p'$ e $\{\!\{p''\}\!\} \equiv p''$. Anche qui per la definizione di forma normale e in base agli assiomi di congruenza strutturale per il Multi-CCS che $\{\!\{p'\}\!\} + \{\!\{p''\}\!\}$ è già in forma normale.
- $p = p' \,|\, p''$ Supponiamo per ipotesi induttiva che $\{\!\{p'\}\!\}$ e $\{\!\{p''\}\!\}$ siano due processi in forma normale tale che: $p' \equiv \{\!\{p'\}\!\}$ e $p'' \equiv \{\!\{p''\}\!\}$; come conseguenza dell'ipotesi induttiva si ha che $p' \,|\, p'' \equiv \{\!\{p'\}\!\} \,|\, \{\!\{p''\}\!\}$. La tesi, quindi l'esistenza della forma normale $\{\!\{p' \,|\, p''\}\!\} \equiv p' \,|\, p''$, segue dagli asserti I e II del Teorema 1.

$p = (\nu\, a)p'$. Supponiamo per ipotesi induttiva che $\{\!\{p'\}\!\}$ sia il processo in forma normale tale che: $\{\!\{p'\}\!\} \equiv p'$, anche qui, come conseguenza dell'ipotesi induttiva abbiamo che $(\nu\, a)p' \equiv (\nu\, a)\{\!\{p'\}\!\}$. Allora $(\nu\, a)\{\!\{p'\}\!\}$ è già in forma normale (tutte le restizioni sono poste all'esterno).

□

Teorema 3 (Unicità della forma normale)**.** *La forma normale di un processo $p \in \mathcal{P}_{Multi-CCS}$ è unica a meno di α-conversione e dell'ordine delle restrizioni.*

Dimostrazione. Segue dalla definizione di forma normale e dal Teorema 2 □

Punto essenziale della dimostrazione che verrà, è dimostrare che due processi p e q sono congruenti, rispetto agli assiomi di congruenza, se e solo se è possibile raggiungere, tramite la navigazione con l'operatore $\overset{-1*}{\rightharpoonup}_{ca}$, la forma normale del processo q a partire dalla forma normale del processo p. Ci siamo resi conto che implementare questo tipo di semantica operazionale con la presenza della quinta regola di congruenza strutturale, questo lemma non era vero e spieghiamo subilto il motivo con un esempio:

Esempio 3. *Consideriamo la costante A definita come segue: $A \overset{def}{=} b.A$ e consideriamo i termini A e $b.A$; i due sono congruenti, ma $\{\!\{A\}\!\} = b.A$ e $\{\!\{b.A\}\!\} = b.b.A$ non c'è modo di navigare dal primo al secondo tramite l'operatore $\overset{-1*}{\rightharpoonup}_{ca}$. Una soluzione, drastica ma funzionante da un punto di vista implementativo è di modificare la sintassi del Multi-CCS, in modo che i processi "legali" siano quei processi p tale che $p = pre(p)$. Ancora, modificare la definizione di semantica operazionale, mettendo il "pre" all'assioma del prefisso. In questo modo si può eliminare la quinta regola, quella della costante, dalle regole di congruenza strutturale.*

Diamo vita quindi a due Multi-CCS, una è quella originale presentata nel Capitolo 2 (Multi-CCS$_{\equiv_5}$), l'altra è quella usata come riferimento nell'implementazione (Multi-CCS$_{\equiv_4}$).

Lemma 1 (Sia $p \in P_{Multi-CCS_{\equiv_5}}$ e $p' \in P_{Multi-CCS_{\equiv_4}}$; abbiamo che $p \sim p'$)**.**

Lemma 2 ($p \equiv q$ sse $\{\!\{p\}\!\} \stackrel{-1*}{\rightharpoonup}_{ca} \{\!\{q\}\!\}$)**.**

Dimostrazione. A Questo punto, il fatto è reso evidente dalla presentazione riservata, ad inizio paragrafo, alla procedura in questione e dalla definizione di forma normale: la procedura $\stackrel{-1*}{\rightharpoonup}_{ca}$ applicata ad un processo in forma normale, quindi passando in ingresso un processo a cui si è applicato fino a quando era possibile le regole di allargamento dello scope e di α-conversione e la procedura "*pre*", definisce in uscita un insieme di processi che rappresentano tutte le possibili associazioni e commutazioni del parallelo. □

3.2 Trasformazione delle SOS originali

Dal Capitolo 2 e alla luce di queste nostre considerazioni, proponiamo in Tabella 3.2 le nuove regole di semantica operazionale del Multi-CCS; il nostro scopo in questa sezione è di renderle più "implementabili"[2].

Trasformiamo quindi le regole nel modo rappresentato in Tabella 3.4[3]. Come si può notare, le nuove regole si differenziano da quelle originali per la loro disposizione su livelli differenti. Mentre per le regole originali tutte le regole sono allo stesso livello, con la modifica si ha che la regola ($Cong_B$) è l'unica regola al primo livello e richiama un insieme di regole SOS che sono tutte le regole originali, a meno della regola di congruenza.

La relazione di "commutativa associativa destra" ($\stackrel{-1}{\rightharpoonup}_{ca}$), utilizzata nella Tabella 3.4, è definita come l'inverso della relazione di "commutativa associativa sinistra" utilizzata per la costruzione della forma normale -chiaramente la $\stackrel{-1*}{\rightharpoonup}_{ca}$ è la chiusura transitiva e riflessiva dell'operatore ad un passo-; essa "costruisce" tutte le possibili associazioni e commutazioni dell'operatore di composizione parallela definita dalla prima e seconda equazione strut-

[2]Discuteremo dell'implementazione e dei problemi incontrati nel Capitolo 4, quando parleremo di com'è stato implementato l'interprete

[3]Le regole sync restano le stesse di quelle riportate nel capitolo 2.

(Pref)	$\mu.p \xrightarrow{\mu} pre(p)$	(S-Pref$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{\underline{\tau}.p \xrightarrow{\sigma} p'}$	
(S-Pref$_2$)	$\dfrac{p \xrightarrow{\tau} p'}{\underline{\alpha}.p \xrightarrow{\alpha} p'}$	(S-Pref$_3$)	$\dfrac{q \xrightarrow{\sigma} q' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\alpha\sigma} p'}$	
(Sum$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{p + q \xrightarrow{\sigma} p'}$	(Sum$_2$)	$\dfrac{q \xrightarrow{\sigma} q'}{p + q \xrightarrow{\sigma} q'}$	
(Par$_1$)	$\dfrac{p \xrightarrow{\sigma} p'}{p \,\vert\, q \xrightarrow{\sigma} p' \,\vert\, q}$	(S-Com)	$\dfrac{q \xrightarrow{\sigma_2} q' \qquad p \xrightarrow{\sigma_1} p'}{p \,\vert\, q \xrightarrow{\sigma} p' \,\vert\, q'}$	$Sync(\sigma_1, \sigma_2, \sigma)$
(Par$_2$)	$\dfrac{q \xrightarrow{\sigma} q'}{p \,\vert\, q \xrightarrow{\sigma} p \,\vert\, q'}$	(S-Res)	$\dfrac{p \xrightarrow{\sigma} p'}{(\nu\, a)p \xrightarrow{\sigma} (\nu\, a)p'}$	$a, \overline{a} \notin n(\sigma)$
(Cong)	$\dfrac{p \equiv p' \xrightarrow{\sigma} q' \equiv q}{p \xrightarrow{\sigma} q}$			

Tabella 3.2: SOS A: Assiomi di inferenza originali per il Muli-CCS.

E1	associatività	$(p \,\vert\, (q \,\vert\, r)) \equiv ((p \,\vert\, q) \,\vert\, r)$	
E2	commutatività	$p \,\vert\, q \equiv q \,\vert\, p$	
E3$_1$	Scope1	$(\nu\, a)(p \,\vert\, q) \equiv (p \,\vert\, (\nu\, a)q)$	se a non è libera in p
E3$_2$	Scope2	$(\nu\, a)(p \,\vert\, q) \equiv ((\nu\, a)p \,\vert\, q)$	se a non è libera in q
E4	α-conversione	$(\nu\, a)p \equiv (\nu\, b)p\{b/a\}$	se $b \in n(x)$

Tabella 3.3: Regole per la congruenza strutturale.

turale per il Multi-CCS, riscrivendo i processi da un'associazione a sinistra a un'associazione a destra.

Se come parametro alla relazione associazione destra applichiamo la forma normale del processo (alla luce dei risultati conseguenti dalla definizione di forma normale), abbiamo che la relazione in questione "naviga" tutta la classe associativa e commutativa del processo entrante.

$$(\text{Cong}_B)\quad \frac{\{\!\{p\}\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p' \qquad p' \overset{\sigma}{\rhd} q' \qquad \{\!\{q\}\!\} \stackrel{-1*}{\rightharpoonup}_{ca} q'}{p \overset{\sigma}{\mapsto} q}$$

(Pref_B)	$\mu.p \overset{\mu}{\rhd} pre(p)$	(S-Pref_{1B})	$\dfrac{p \overset{\sigma}{\rhd} p'}{\underline{\tau}.p \overset{\sigma}{\rhd} p'}$	
(S-Pref_{2B})	$\dfrac{p \overset{\tau}{\rhd} p'}{\underline{\alpha}.p \overset{\alpha}{\rhd} p'}$	(S-Pref_{3B})	$\dfrac{q \overset{\overline{\alpha}}{\rhd} q' \qquad \sigma \neq \tau}{\underline{\alpha}.p \overset{\alpha\sigma}{\rhd} p'}$	
(Sum_{1B})	$\dfrac{p \overset{\sigma}{\rhd} p'}{p + q \overset{\sigma}{\rhd} p'}$	(Sum_{2B})	$\dfrac{q \overset{\sigma}{\rhd} q'}{p + q \overset{\sigma}{\rhd} q'}$	
(Par_{1B})	$\dfrac{p \overset{\sigma}{\rhd} p'}{p \mid q \overset{\sigma}{\rhd} p' \mid q}$	(S-Com_B)	$\dfrac{q \overset{\sigma_2}{\rhd} q' \qquad p \overset{\sigma_1}{\rhd} p'}{p \mid q \overset{\sigma}{\rhd} p' \mid q'}$	$Sync(\sigma_1, \sigma_2, \sigma)$
(Par_{2B})	$\dfrac{q \overset{\sigma}{\rhd} q'}{p \mid q \overset{\sigma}{\rhd} p \mid q'}$	(S-Res_B)	$\dfrac{p \overset{\sigma}{\rhd} p'}{(\nu\, a)p \overset{\sigma}{\rhd} (\nu\, a)p'}$	$a, \overline{a} \notin n(\sigma)$

$$\frac{}{\mathbb{C}\{((P \mid Q) \mid R)\} \stackrel{-1}{\rightharpoonup}_{as} \mathbb{C}\{(P \mid (Q \mid R))\}}$$

Tabella 3.4: SOS B: Regole di inferenza modificati per il Muli-CCS

.

Esempio 4. *Dato il processo* $P = (\nu\, a)(a \mid ((b \mid c) \mid d))$ *la relazione* $\stackrel{-1*}{\rightharpoonup}_{ca}$ *costruisce l'insieme* C *in Figura 3.1 dei processi ad esso congruenti secondo la prima equazione strutturale, effettuando tutte le possibili associazioni rispetto all'operatore di composizione parallela, in più, per ogni processo nell'insieme* C*, l'operatore* $\stackrel{-1*}{\rightharpoonup}_{ca}$ *effettua tutte le possibile commutazioni tra i sotto-processi secondo la seconda equazione strutturale. Quindi* $P \stackrel{-1*}{\rightharpoonup}_{ca} P'$ *associa a* P' *tutti i risultati delle due riscritture.*

$$C = \left\{ \begin{array}{l} (\nu\, a)(a \,|\, (b \,|\, (c \,|\, d))); \\ (\nu\, a)(a \,|\, ((b \,|\, c) \,|\, d)); \\ (\nu\, a)((a \,|\, b) \,|\, (c \,|\, d)); \\ (\nu\, a)((a \,|\, (b \,|\, c)) \,|\, d); \\ (\nu\, a)(((a \,|\, b) \,|\, c) \,|\, d) \end{array} \right\}$$

Figura 3.1: Insieme dei processi associativi a P.

Teorema 4 ($P \xrightarrow{\alpha} Q$ sse $P \overset{\alpha}{\mapsto} Q$)**.** *Le regole SOS originali del Multi-CCS (A) e quelle modificate (B) sono equivalenti.*

Dimostrazione. ($\Rightarrow$) Per induzione sull'albero di derivazione di $p \xrightarrow{\alpha} q$:

- caso base: $\mu.p \xrightarrow{\mu} p$

$$\frac{\{\!\{\mu.p\}\!\} \overset{-1*}{\rightharpoonup}_{ca} \mu.p \qquad \mu.p \overset{\mu}{\rhd} p \qquad \{\!\{p\}\!\} \overset{-1*}{\rightharpoonup}_{ca} p}{\mu.p \overset{\mu}{\mapsto} p}$$

- casi induttivi:
  - caso $\dfrac{p \xrightarrow{\sigma} p'}{\underline{\tau}.p \xrightarrow{\sigma} p'}$:
    IH $= p \overset{\sigma}{\mapsto} p'$

$$\frac{\{\!\{\underline{\tau}.p\}\!\} \overset{-1*}{\rightharpoonup}_{ca} \underline{\tau}.p \qquad \dfrac{p \overset{\sigma}{\rhd} p'}{\underline{\tau}.p \overset{\sigma}{\rhd} p'} \qquad \{\!\{p'\}\!\} \overset{-1*}{\rightharpoonup}_{ca} p'}{\underline{\tau}.p \overset{\sigma}{\mapsto} p'}$$

  - caso $\dfrac{p \xrightarrow{\tau} p'}{\underline{\alpha}.p \xrightarrow{\alpha} p'}$
    IH $= p \overset{\tau}{\mapsto} p'$

$$\frac{\{\!\{\underline{\alpha}.p\}\!\} \overset{-1*}{\rightharpoonup}_{ca} \underline{\alpha}.p \qquad \dfrac{p \overset{\tau}{\rhd} p'}{\underline{\alpha}.p \overset{\sigma}{\rhd} p'} \qquad \{\!\{p'\}\!\} \overset{-1*}{\rightharpoonup}_{ca} p'}{\underline{\alpha}.p \overset{\sigma}{\mapsto} p'}$$

  - caso $\dfrac{p \xrightarrow{\sigma} p' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\alpha\sigma} p'}$
    IH $= p \overset{\sigma}{\mapsto} p'$

$$\frac{\{\!|\underline{\alpha}.p|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} \underline{\alpha}.p \qquad \dfrac{p \overset{\sigma}{\rhd} p' \qquad \sigma \neq \tau}{\underline{\alpha}.p \overset{\alpha\sigma}{\rhd} p'} \qquad \{\!|p'|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p'}{\underline{\alpha}.p \overset{\alpha\sigma}{\mapsto} p'}$$

– caso $\dfrac{p \overset{\sigma}{\to} p'}{p + q \overset{\sigma}{\to} p'}$

IH $= p \overset{\sigma}{\mapsto} p'$

$$\frac{\{\!|p + q|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p + q \qquad \dfrac{p \overset{\sigma}{\rhd} p'}{p + q \overset{\sigma}{\rhd} p'} \qquad \{\!|p'|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p'}{p + q \overset{\sigma}{\mapsto} p'}$$

– caso $\dfrac{q \overset{\sigma}{\to} q'}{p + q \overset{\sigma}{\to} q'}$

IH $= p \overset{\sigma}{\mapsto} p'$

$$\frac{\{\!|p + q|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p + q \qquad \dfrac{q \overset{\sigma}{\rhd} q'}{p + q \overset{\sigma}{\rhd} q'} \qquad \{\!|q'|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} q'}{p + q \overset{\sigma}{\mapsto} q'}$$

– caso $\dfrac{p \overset{\sigma}{\to} p'}{p \,|\, q \overset{\sigma}{\to} p' \,|\, q}$

IH $= p \overset{\sigma}{\mapsto} p'$

$$\frac{\{\!|p \,|\, q|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p \,|\, q \qquad \dfrac{p \overset{\sigma}{\rhd} p'}{p \,|\, q \overset{\sigma}{\rhd} p' \,|\, q} \qquad \{\!|p' \,|\, q|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p' \,|\, q}{p \,|\, q \overset{\sigma}{\mapsto} p' \,|\, q}$$

– caso $\dfrac{q \overset{\sigma}{\to} q'}{p \,|\, q \overset{\sigma}{\to} p \,|\, q'}$

IH $= q \overset{\sigma}{\mapsto} q'$

$$\frac{\{\!|p \,|\, q|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p \,|\, q \qquad \dfrac{q \overset{\sigma}{\rhd} q'}{p \,|\, q \overset{\sigma}{\rhd} p \,|\, q'} \qquad \{\!|p \,|\, q'|\!\} \stackrel{-1*}{\rightharpoonup}_{ca} p \,|\, q'}{p \,|\, q \overset{\sigma}{\mapsto} p \,|\, q'}$$

– caso $\dfrac{p \overset{\sigma_1}{\to} p' \qquad q \overset{\sigma_2}{\to} q'}{p \,|\, q \overset{\sigma}{\to} p' \,|\, q'}$

$\text{IH} = p \overset{\sigma_1}{\mapsto} p',\ q \overset{\sigma_2}{\mapsto} q'$

$$\dfrac{\{\!|p \,|\, q|\!\} \overset{-1*}{\rightharpoonup}_{ca} p \,|\, q \qquad \dfrac{p \overset{\sigma_1}{\rhd} p' \qquad q \overset{\sigma_2}{\rhd} q'}{p \,|\, q \overset{\sigma}{\rhd} p' \,|\, q'} \qquad \{\!|p' \,|\, q'|\!\} \overset{-1*}{\rightharpoonup}_{ca} p' \,|\, q'}{p \,|\, q \overset{\sigma}{\mapsto} p' \,|\, q'}$$

– caso $\dfrac{p \overset{\sigma}{\to} p'}{(\nu\, a)p \overset{\sigma}{\to} (\nu\, a)p'}\ a, \overline{a} \notin n(\sigma)$

$\text{IH} = p \overset{\sigma}{\mapsto} p'$

$$\dfrac{\{\!|(\nu\, a)p|\!\} \overset{-1*}{\rightharpoonup}_{ca} (\nu\, a)p \qquad \dfrac{p \overset{\sigma}{\rhd} p'}{(\nu\, a)p \overset{\sigma}{\rhd} (\nu\, a)p'} \qquad \{\!|(\nu\, a)p'|\!\} \overset{-1*}{\rightharpoonup}_{ca} (\nu\, a)p'}{(\nu\, a)p \overset{\sigma}{\mapsto} (\nu\, a)p'}$$

– caso $\dfrac{p \equiv p' \overset{\sigma}{\to} q' \equiv q}{p \overset{\sigma}{\to} q}$

$\text{IH} = p' \overset{\sigma}{\mapsto} q'$

$$\dfrac{\{\!|p|\!\} \overset{-1*}{\rightharpoonup}_{ca} p' \qquad p' \overset{\sigma}{\rhd} q' \qquad \{\!|q|\!\} \overset{-1*}{\rightharpoonup}_{ca} q'}{p \overset{\sigma}{\mapsto} q}$$

Spieghiamo l'unico caso che merita una spiegazione. Come si può notare dalla nuova definizione di regole SOS, non è possibile richiamare più volte la regola $(Cong_B)$, quindi una derivazione come la seguente, effettuata con le regole dell'insieme A, sembrerebbe non possibile con le regole dell'insieme B;

$$\dfrac{p \equiv p' \qquad \dfrac{p' \equiv p'' \qquad \dfrac{\vdots}{p'' \overset{\sigma}{\to} q''}\,\text{cong} \qquad q'' \equiv q'}{p' \overset{\sigma}{\to} q'} \qquad q' \equiv q}{p \overset{\sigma}{\to} q}$$

si ha quindi che: $p \equiv p' \equiv p'' \equiv \ldots \equiv p^n$.

La regola $Cong_B$ richiama la procedura $\overset{-1*}{\rightharpoonup}_{as}$, che, come abbiamo avuto modo di dire, è in grado di navigare tutta la classe di associatività della forma normale del processo passato come aromento. Quindi per completare

la nostra dimostrazione sull'equivalenza delle Regole A e Regole B, dobbiamo mostrare che la procedura $\rightharpoonup_{ca}^{-1}$ applicata alla forma normale del processo in esecuzione, corrisponde alla ricorsione della $(Cong)$ dell'insieme delle Regole A.

La stessa transizione con le regole dell'insieme B coicide con la definizione stessa di $\mapsto$.

$$\frac{\{\!\{p\}\!\} \rightharpoonup_{ca}^{-1*} p^n \qquad p^n \overset{\sigma}{\rhd} q^n \qquad \{\!\{q\}\!\} \rightharpoonup_{ca}^{-1*} q^n}{p \overset{\sigma}{\mapsto} q}$$

($\Leftarrow$) L'inverso si dimostra per induzione sull'albero di derivazione di $p \overset{\alpha}{\mapsto} q$

- caso base: $\dfrac{\{\!\{\mu.p\}\!\} \rightharpoonup_{ca}^{-1*} \mu.p_1 \qquad \mu.p_1 \overset{\mu}{\rhd} p_1' \qquad \{\!\{p'\}\!\} \rightharpoonup_{ca}^{-1*} p_1'}{\mu.p \overset{\sigma}{\mapsto} p'}$

  $$\frac{p \equiv p_1 \qquad p \overset{\mu}{\to} p_1' \qquad p_1' \equiv p'}{\mu.p \overset{\mu}{\to} p'}$$

  la corrispondenza $p \equiv p_1$ se $\{\!\{\mu.p\}\!\} \rightharpoonup_{ca}^{-1*} \mu.p_1$ e $p_1' \equiv p'$ se $\{\!\{p'\}\!\} \rightharpoonup_{ca}^{-1*} p_1'$ è dimostrata dal Lemma 2, e $p \overset{\mu}{\to} p_1'$ è un assioma della semantica originale del Multi-CCS

- caso induttivo:

  - caso $\dfrac{\{\!\{\underline{\tau}.p\}\!\} \rightharpoonup_{ca}^{-1*} pre(\underline{\tau}.p_1) \qquad pre(\underline{\tau}.p_1) \overset{\sigma}{\rhd} p_1' \qquad \{\!\{p_1'\}\!\} \rightharpoonup_{ca}^{-1*} p'}{\underline{\tau}.p \overset{\sigma}{\mapsto} p'}$

    IH $= p_1 \overset{\sigma}{\to} p_1'$

    $$\frac{\underline{\tau}.p \equiv \underline{\tau}.p_1 \qquad \dfrac{p_1 \overset{\sigma}{\to} p_1'}{\underline{\tau}.p_1 \overset{\sigma}{\to} \underline{\tau}.p_1'} \qquad p_1' \equiv p'}{\underline{\tau}.p \overset{\sigma}{\to} p'}$$

    Anche in questo caso la corrispondenza $\underline{\tau}.p \equiv \underline{\tau}.p_1$ se $\{\!\{\underline{\tau}.p\}\!\} \rightharpoonup_{ca}^{-1*} pre(\underline{\tau}.p_1)$ e $p_1' \equiv p'$ se $\{\!\{p_1'\}\!\} \rightharpoonup_{ca}^{-1*} p'$ è assicurata dal Lemma 2, mentre $p_1 \overset{\sigma}{\to} p_1'$ dall'ipotesi induttiva.

□

Esempio 5. *Riaffrontiamo il caso della sincronizzazione Multi-party con l'utilizzo della regola* $(Cong_B)$*. Sia quindi* $P = (\nu\, a)(\underline{a}.a.p' \mid (\nu\, b)(\overline{a}.q' \mid \overline{a}.b.r'))$ *e* $P' = (\nu\, a)(p' \mid (\nu\, b)(q' \mid b.r'))$*. Abbiamo da dimostrare la transizione:*

$$P \overset{\tau}{\mapsto} P'.$$

Costruiamo la forma normale del processo P *applicando, come da definizione, prima l'allargamento dello scope e l'*α*-conversione, poi successivamente l'associatività sinistra rispetto all'operatore parallelo. Abbiamo che il nome di canale "a" viene sostituito con il nome di canale "*$\#a_2$*" e il nome di canale "b" con il nome di canale "*$\#b_1$*".*

$$\begin{aligned}\{\!\{P\}\!\} &= (\nu\, a, b)((\underline{a}.a.p' \mid \overline{a}.q') \mid \overline{a}.b.r')\{a/\#a_2\}\{b/\#b_1\} \text{ quindi:}\\ &(\nu\, \#a_2, \#b_1)((\underline{\#a_2}.\#a_2.p' \mid \overline{\#a_2}.q') \mid \overline{\#a_2}.\#b_1.r')\}\end{aligned}$$

Definiamo la classe associativa di $\{\!\{P\}\!\}$*:*

$$[\{\!\{P\}\!\}]_{Multi-CCS} = \left\{ \begin{array}{l} \{\!\{P_1\}\!\} = (\nu\, \#a_2, \#b_1)((\underline{\#a_2}.\#a_2.p' \mid \overline{\#a_2}.q') \mid \overline{\#a_2}.\#b_1.r') \\ \{\!\{P_2\}\!\} = (\nu\, \#a_2, \#b_1)(\underline{\#a_2}.\#a_2.p' \mid (\overline{\#a_2}.q' \mid \overline{\#a_2}.\#b_1.r')) \end{array} \right\}$$

Costruiamo la forma normale di P'*:*

$$\begin{aligned}\{\!\{P'\}\!\} &= (\nu\, a, b)((p' \mid q') \mid b.r')\{a/\#a_2\}\{b/\#b_1\} \text{ quindi:}\\ &(\nu\, \#a_2, \#b_1)((p' \mid q') \mid \#b_1.r').\end{aligned}$$

Definiamo la classe associativa di $\{\!\{P'\}\!\}$*:*

$$[\{\!\{P'\}\!\}]_{Multi-CCS} = \left\{ \begin{array}{l} \{\!\{P'_1\}\!\} = (\nu\, \#a_2, \#b_1)((p' \mid q') \mid \#b_1.r') \\ \{\!\{P'_2\}\!\} = (\nu\, \#a_2, \#b_1)(p' \mid (q' \mid \#b_1.r')) \end{array} \right\}$$

Con le regole A abbiamo che l'albero che dimostra la transizione richiesta è molto simile all'albero mostrato a Pagine 24

Con le regole B invece la derivazione è la seguente:

$$\dfrac{\{\!\{P\}\!\} \dashrightarrow_{as}^{-1} \{\!\{P_2\}\!\} \qquad \dfrac{\Psi}{\{\!\{P_2\}\!\} \overset{\tau}{\rhd} \{\!\{P'_2\}\!\}} \qquad \{\!\{P'\}\!\} \dashrightarrow_{as}^{-1} \{\!\{P'_2\}\!\}}{(\nu\, a)(\underline{a}.a.p' \mid (\nu\, b)(\overline{a}.q' \mid \overline{a}.b.r')) \overset{\tau}{\mapsto} (\nu\, a)(p' \mid (\nu\, b)(q' \mid b.r'))}$$

con Ψ *albero composto come segue:*

$$\text{(Res)I}\ \frac{\text{(Com)II}\ \dfrac{\text{(Com)III}\ \dfrac{\textit{(s\_Pref)}\ \dfrac{a.p' \overset{a}{\rhd} p'}{}}{\underline{\#a_2}.a.p' \overset{\underline{\#a_2}a}{\rhd} p' \qquad \overline{\#a_2}.q' \overset{\overline{\#a_2}}{\rhd} q'}}{(\underline{\#a_2}.a.p' \mid \overline{\#a_2}.q') \overset{\#a_2}{\rhd} (p' \mid q') \qquad \overline{\#a_2}.\#b_1.r' \overset{\overline{\#a_2}}{\rhd} \#b_1.r'}}{((\underline{\#a_2}.a.p' \mid \overline{\#a_2}.q') \mid \overline{\#a_2}.\#b_1.r') \overset{\tau}{\rhd} ((p' \mid q') \mid \#b_1.r')}}{(\nu\, \#a_2, \#b_1)((\underline{\#a_2}.\#a_2.p' \mid \overline{\#a_2}.q') \mid \overline{\#a_2}.\#b_1.r') \overset{\tau}{\rhd} (\nu\, \#a_2, \#b_1)((p' \mid q') \mid \#b_1.r')}$$

Nulla di interessante c'è da dire sulle regole *sync* che vengono riproposte identiche nell'insieme della regole B.

Capitolo 4

Interprete per Multi-CCS: Multi-muCCS

In questo capitolo viene presentato un interprete per Multi-CCS; Multi-muCCS ottenuto come un'estensione di muCCS -interprete CCS- creato da *Andrea Simonetto*, *Paolo Perfetti*, *Odeta Qorri*, la cui documentazione e sorgenti sono disponibili sulla pagina personale di Andrea Simonetto[**?**].

Illustriamo in questa sede gli aspetti fondamentali del Multi-muCCS, ovvero: le scelte fatte sull'implementazione della sintassi e della semantica del Multi-CCS, gli aspetti teorici ad essi legati, e la sintassi del Multi-muCCS stesso.

4.1 Multi-muCCS

Presentazione del Multi-muCCS

Multi-muCCS attualmente è solo per sistemi linux. Si avvia da terminale eseguendo il binario *muccs* prensente nella cartella principale del progetto dopo la compilazione eseguita con il comando *make*. Dal terminale di dialogo è possibile interagire con una serie di comandi in grado di definire processi ed eseguire le funzionalita messe a disposizione. L' interprete Multi-muCCS soddisfa i seguenti requisiti (le funzionalità):

- definisce un'appropriata struttura per stati e transizioni;

- fornisce un'appropriata rappresentazione delle regole SOS sia del CCS che del Multi-CCS;
- verifica se una transizione è derivabile dalle regole SOS;
- dato uno stato/processo calcola tutte le transizioni esistenti a partire da quello stato;
- dati due stati/processi $P1$ e $P2$ verifica se $P2$ può essere raggiunto da $P1$;
- dato uno stato/processo P determina il suo LTS associato.

I comandi invece sono:

help : mostra i comandi disponibili;

agent : inserisce una definizione CCS;

graphlts : dato uno stato/processo crea un file .svg con l'LTS associato;

lts : Dato uno stato/processo determina il suo LTS associato;

print : mostra l'ambiente corrente;

quit : esce dal programma;

reach : dati due stati, verifica se uno stato puó raggiungere l'altro;

test : testa una transizione;

trans : mostra le transizioni in uscta da un dato processo/stato;

In accordo con la definizione della sintassi del Multi-CCS [GV10, Gor12], l'insieme $\mathcal{P}$ delle espressioni valide è definibile in Multi-muCCS con la seguente grammatica data in formalismo BNF[1]:

[1] Backus-Naur-Form è una metasintassi, ovvero un formalismo attraverso il quale è possibile descrivere la sintassi di linguaggi formali. In termini formali, la BNF può essere vista come uno strumento per descrivere grammatiche libere dal contesto.

- $p ::= 0 \mid \mu.q \mid \_\mu.q \mid p + p$ processi sequenziali
- $q ::= p \mid q|q \mid q \setminus \{L\} \mid C$ processi

4.2 Aspetti teorici e scelte implementative

Illustriamo ora gli aspetti teorici e le scelte implementative che caratterizzano il Multi-muCCS. Analiziamo, per dare inizio a questa fase, lo schema principale del progetto illustrato in Figura 4.1

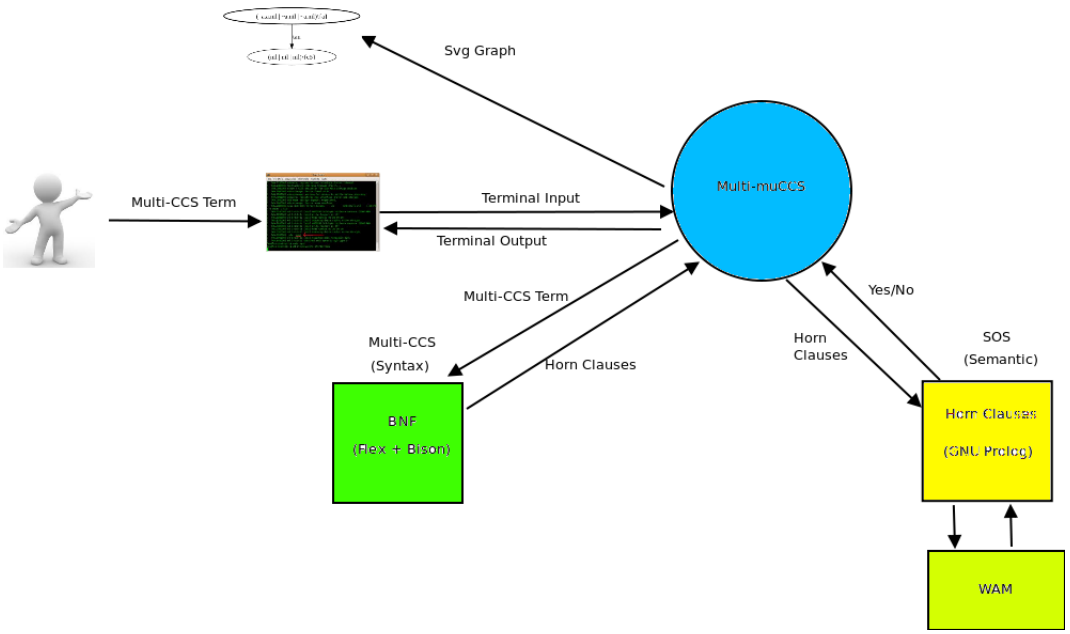

Figura 4.1: Schema principale di muCCS e Multi-muCCS

Lo schema evidenzia i due concetti principali per ogni linguaggio: la Sintassi e la Semantica. Per quanto concerne la prima, vengono usati Flex e Bison per la generazione di un parser per un linguaggio Context Free. La semantica invece viene introdotta attraverso l'implementazione a Clausole di Horn (vedi Pagina 77) delle regole SOS, e utilizzeremo per questo scopo GNU-

Prolog come motore inferenziale [2]. L'uso del linguaggio di programmazione C aiuta ad attuare le interrogazioni e a costruire l'output finale; è di fatto il ponte che collega i due mondi - sintassi e semantica- nel nostro interprete, realizzando l'interfaccia per l'utente.

Il lavoro dell'interprete: una volta immesso il comando il parser inizia a determinare la sua struttura grammaticale generando il rispettivo albero sintattico. A questo punto il parser traduce i token riconosciuti in proposizioni Prolog, richiamando quindi il motore inferenziale per la risoluzione della richiesta. Analizzando la grammatica formale del Multi-CCS ci si rende conto che per l'interpretazione di un funtore non c'è bisogno di analizzare fino in fondo il sottoalbero, questo facilita il lavoro del parser che può, a questo punto, unire le due fasi di creazione dell'albero sintattico e della traduzione dei token di procedure prolog. Questo tipo di parser si chiama "parser guidato dalla sintassi"

Analiziamo ora l'implementazione della semantica del Multi-CCS. Dalla definizione data nel Paragrafo 2.5, una semantica in termini di LTS è una tripla $(\mathcal{P}, \mathcal{A}, \rightarrow)$ con $\mathcal{P}$ l'insieme dei processi "legali", $\mathcal{A}$ l'insieme delle etichette e $\rightarrow$ la più piccola relazione in grado di soddisfare le regole SOS in Tabella 2.2. Quindi per determinare l'LTS di un processo Multi-CCS non dobbiamo fare altro che creare la relazione di transizione "$\rightarrow$", implementando le regole SOS. Per questo scopo usiamo come linguaggio di programmazione il Prolog. Prolog, il cui compilatore GNU-Prolog è completo e facile da usare, ha diversi vantaggi che ne motivano l'utilizzo nell'implementazione del cuore dell'interprete: manipola relazioni, incorpora l'essenziale meccanismo di backtracking[3], la sua sintassi e semantica sono basate su clausole di Horn, il modo più naturale per esprimere le regole SOS.

[2] Il motore inferenziale è un programma costituito da un interprete che decide quale regola applicare per poter aumentare la base di conoscenza e da uno schedulatore che organizza le regole da sviluppare e il loro ordine di esecuzione.

[3] Il meccanismo per cercare multiple soluzioni, utile per gestire il non-determinismo tipoco dei linguaggi concorrenti.

4.3 Implementazione del Multi-muCCS

4.3.1 Implementazione della sintassi

Le regole lessicali sono definite dalla configurazione del file di Flex (multi-ccs.l). Di seguito mostriamo le principali regole definite per riconoscere i token della sintassi:

```
1 [A-Z][a-zA-Z0-9]* {return PID; }
  [a-z][a-zA-Z0-9]*   {return OBS_ACT; }
3 0|nil      { return NIL; }
  tau      { return TAU; }
5 \~       { return OP_OUT; }
  \.       { return OP_PREFIX; }
7 \_       { return OP_ST_PREFIX; }
  \+       { return OP_SUM; }
9 \|       { return OP_COM; }
  \\       { return OP_RESTRICT; }
```

I termini Prolog sono costruiti sfruttando le azioni corrispondenti ad ogni regola presente nel file Bison durante il processo di parsing, utilizzando le API per il linguaggio C del GNU-Prolog.

L'insieme $\mathcal{P}$ viene riconosciuto dal Multi-muCCS grazie all'utilizzo combinato di Flex (GNU lexer generator) e Bison (GNU parser generator). Il primo riconosce l'input e genera i token utilizzati dal secondo per individuare le frasi ben formate, in base alla definizione di grammatica data sopra. Questa grammatica libera da contesto è usata per definire la sintassi di un linguaggio di programmazione, utilizzando due insiemi di regole: regole lessicali e regole sintattiche.

Le regole sintattiche sono definite (usando i token provenienti da Flex) nel file di configurazione di Bison (in multi-ccs.y). Definiamo uno speudo codice che spiga alcuni elementi della grammatica Multi-CCS:

```
  ProcDef ::= PID OP_DEF Proc
2
```

```
Proc   ::= (Proc) | NIL | PID |TAU OP_PREFIX Proc | OBS_ACT.proc
    | OP_OUT OBS_ACT Proc | Proc OP_SUM Proc | Proc OP_COM Proc |
     OP_ST_PREFIX OBS_ACT Proc | Proc OP_RESTICT{ObsActList}

ObsActList   ::= OBS_ACT | OBS_ACT, ObsActList

Rel    ::=    OBS_ACT SLASH OBS_ACT

RelList ::= Rel | Rel, RelList
```

4.3.2 Implementazione della semantica

Nel file "*sos.pl*" vengono implementate in Prolog le regole SOS, concentriamoci su ognuna di queste regole. La traduzione delle regole (dal linguaggio matematico alla sintassi Prolog) è molto facile ed immediata, solo la regola (*Cong*) richiederà un paragrafo a parte.

- (*Pref*)

$$(\mathsf{Pref})\ \mu.p \xrightarrow{\mu} p$$

  Per emulare questo assioma, viene creato un funtore "*pre*" per esprimere in Prolog il prefisso del Multi-CCS, in questo modo la regola viene tradotta in Prolog nel seguente modo:

```
move_aux(pre(Mu, P), P, [Mu]).
```

  che significa che il processo $pre(Mu, P)$ può muovere in P con lista di mosse $[Mu]$.

- (*s_Pref*)

$$(\mathsf{s\_Pref})\ \frac{p \xrightarrow{\sigma} p'}{\underline{\mu}.p \xrightarrow{\mu\sigma} p'}$$

  In questa caso invece viene creato il funtore "*strong*".

```
1 move_aux(strong(Mu,P),P_1,L):-move_aux(P,P_1,Sigma),Sigma
      \==[],L=[Mu|Sigma].
```

Quindi $strong(Mu, P)$ muove in P_1 con lista di azioni L se P muove verso P_1 con mossa non nulla $Sigma$ e la lista L è composta dalle azioni $Sigma$ e Mu

- (Sum)

$$(\mathsf{Sum}_l)\ \frac{p \xrightarrow{\sigma} p'}{p + q \xrightarrow{\sigma} p'} \qquad (\mathsf{Sum}_r)\ \frac{q \xrightarrow{\sigma} q'}{p + q \xrightarrow{\sigma} q'}$$

Il funtore in questo caso è "*sum*". Abbiamo bisogno di due procedure per tradurre le due regole.

```
1 move_aux(sum(P,_),P_1,Sigma):-move_aux(P,P_1,Sigma).

3 move_aux(sum(_,Q),Q_1,Sigma):-move_aux(Q,Q_1,Sigma).
```

Questa regola esprime la possibilità di scegliere la prosecuzione della computazione.

- (Com)

$$(\mathsf{Com})\ \frac{p \xrightarrow{\sigma_1} p' \qquad q \xrightarrow{\sigma_2} q'}{p \,|\, q \xrightarrow{\sigma} p' \,|\, q'} \quad Sync(\sigma_1, \sigma_2, \sigma)$$

Il funtore per questa regola e per la successiva è "*com*"

```
1 move_aux(com(P,Q),com(P_1,Q_1),Sigma):-move_aux(P,P_1,Sigma1
      ),move_aux(Q,Q_1,Sigma2),sync(Sigma1,Sigma2,Sigma).
```

Le regole *sync* a margine della procedura vengono implementate come segue:

```
1 % Multi-syncronization
  sync([X], [Y], [tau]) :- comm(X, Y), !.
3 sync([X | R], [Y], R) :- comm(X, Y), R \== [].
  sync([X], [Y | R], R) :- comm(X, Y), R \== [].
5 sync([tau | S1], S2, S) :- sync(S1, S2, S).
  sync(S1, [tau | S2], S) :- sync(S1, S2, S).
7 sync([X | S1], S2, [X | S]) :- sync(S1, S2, S).
  sync(S1, [X | S2], [X | S]) :- sync(S1, S2, S).
9 sync([X | S1], [Y | S2], S) :- comm(X, Y), sync(S1, S2, S).
```

In questo caso il parallelo tra P e Q muove in $P_1 \,|\, Q_1$ con mossa $Sigma$ se esistono rispettivamente due etichette, $Sigma1$ e $Sigma2$, in grado di far muovere i singoli processi, e se $Sigma$, $Sigma1$, $Sigma2$ appartengono alla relazione *sync*.

- (*Par*)

$$(\mathsf{Par}_l)\ \frac{p \xrightarrow{\sigma} p'}{p \,|\, q \xrightarrow{\sigma} p' \,|\, q} \qquad (\mathsf{Par}_r)\ \frac{q \xrightarrow{\sigma} q'}{p \,|\, q \xrightarrow{\sigma} p \,|\, q'}$$

Questa regola descrive la scelta nella composizione parallela, viene implementata come segue.

```
1 move_aux(com(P,Q),com(P_1,Q),Sigma):-move_aux(P,P_1,Sigma).

3 move_aux(com(P,Q),com(P,Q_1),A):-move_aux(Q,Q_1,A).
```

- (*Res*)

$$(\mathsf{Res})\ \frac{p \xrightarrow{\sigma} p'}{(\nu\, a)p \xrightarrow{\sigma} (\nu\, a)p'} \quad a, \overline{a} \notin n(\sigma)$$

Il funtore in questo caso è "*res*."

```
1 move_aux(res(P,A),res(P_1,A),Sigma):-move_aux(P,P_1,Sigma),
      not_in_L(Sigma,A).
```

La procedura quindi implementa la regola rimuovendo l'etichetta *Sigma* dalle legali mosse di P verso P_1.

Implementazione della regola (*cong*)

La sostanziale difficoltà per la costruzione della semantica operazionale del Multi-muCCS è stata l'implementazione della regola (*Cong*) che definisce, come abbiamo già detto, la congruenza tra processi;

ci sono stati diversi tentativi di implementazione della congruenza strutturale; ne esaminiamo velocemente il primo e l'ultimo, i più importanti.

Il primo tentativo è una traduzione "letterale" della definizione matematica di congruenza, quindi:

La traduzione della regola (*cong*) quindi è la seguente:

```
1 move(P,Q,Sigma) :- cong(P,P_1),move(P_1,Q_1,Sigma),cong(Q,Q_1).
```

dove *cong* è una procedura che implementa la congrueza tra i processi nel modo matematico sopra descritto. In questo modo le regole di riflessività e simmetria portavano in loop il sistema.

Esempio 6. $a.p \,|\, a.q = a.q \,|\, a.p = a.p \,|\, a.q = a.q \,|\, a.p \ldots$

L'ultimo e fruttuoso tentativo è stato quello implementare separatamente le equazioni si congruenza strutturale, quindi quella di creare una forma normale. Tutto questo è stato già presentato dimostrato nel Capitolo 3, in questo capitolo non ci resta che presentare il codice PROLOG che implementa la congruenza. La regola SOS principale (l'implementazione della relazione $\mapsto$) è la seguente:

```
1 move(X, Q, Xn, Sigma) :- var(Q), preCostante(X, Xpre),
    fn(Xpre, Xn), predicatoCA(Xn, X1), move_aux(X1, Q1, Sigma),
    fn(Q, Qn) predicatoCA(Qn, Q1).
```

Quindi mostriamo esclusivamente i predicati che implementano l'associatività, la commutatività ("*predicatoCA*") e il nostro "predicatoCostante" che rappresenta l'operatore "*pre*" nel capitolo 3. L'implementazione della forma normale non è, a mio avviso, ritenuta impertante perché effettua esclusivamente una mera manipolazione sintattica del processo.

```
preCostante(nil, nil).
preCostante(P, P1).
preCostante(pre(Sigma, P), pre(Sigma, P)).
preCostante(sum(P, Q), sum(P1, Q1)) :- preCostante(P, P1),
    preCostante(Q, Q1).
preCostante(com(P, Q), com(P1, Q1)) :- preCostante(P, P1),
    preCostante(Q, Q1).
preCostante(res(P, L), res(P1, L)) :- preCostante(P, P1).
preCostante(proc(K), P) :- proc_def(K, P).
```

```
predicatoCA(Tree1, Tree1) :- var(Tree1), !.
predicatoCA(Tree1, res(Tree2, L)) :- Tree1 =.. [res, Tree, L],
    !, predicatoCA(Tree, Tree2).
predicatoCA(Tree1, Tree2) :- flattentree(Tree1, L),
    permutation(L, L1), congtree(L1, Tree2).
```

```
congtree([A], A).
congtree([A, B], com(A, B)).
congtree(LElems, com(PreSubTree, PostSubTree)) :-
  length(LElems, N1), N1 >= 3,
  append(LPre, LPost, LElems),
  length(LPre, N2), N2 >= 1,
  length(LPost, N3), N3 >= 1,
  congtree(LPre, PreSubTree),
  congtree(LPost, PostSubTree).
```

```
1 flattentree(X, L) :- \+ var(X), X = com(A, B), !, flattentree(A,
      L1), flattentree(B, L2), append(L1, L2, L).
  flattentree(X, [X]).
```

Capitolo 5

Multi-CCS linear step: Multi-CCS-ls

La bisimulazione gode di alcune proprietà algebriche attese, ma sfortunatamente essa non è una congruenza in Multi-CCS per la composizione parallela. Al fine di trovare una semantica adatta per la composizione parallela del Multi-CCS, si definisce una semantica operazionale alternativa, dove vengono etichettate multi-insiemi di transizioni eseguibili contemporaneamente. L'ordinaria bisimulazione su questo sistema arricchito è chiamato step bisimulation equivalence.

5.1 Semantica interleaving

Due termini p e q sono interleving bisimilar ($p \sim q$) se esiste una strong bisimulation $\mathcal{R}$ tale che $(p, q) \in \mathcal{R}$. La interleving bisimilar equilavence gode di alcune importanti proprietà algebriche:

Proposizione 1. *Siano $p, q, r \in \mathcal{P}$ processi. Allora valgono le seguenti:*

(1)	$(p + q) + r \sim p + (q + r)$	*(2)*	$p + q \sim q + p$
(3)	$p + 0 \sim p$	*(4)*	$p + p \sim p$
(5)	$p \mid (q \mid r) \sim (p \mid q) \mid r$	*(6)*	$p \mid p \sim p$
(7)	$p \mid 0 \sim p$	*(8)*	$(\nu\, x)(p \mid q) \sim p \mid (\nu\, x)q \quad se\ x \notin fn(p)$
(9)	$(\nu\, x, y)p \sim (\nu\, y, x)p$	*(10)*	$(\nu\, x)p \sim (\nu\, y)p\{y/x\} \quad se\ x \notin fn(p)$
(11)	$(\nu\, x)0 \sim 0$		

Proposizione 2. *Sia $p, q \in \mathcal{P}$ processi Multi-CCS. Se $p \equiv q$ allora $p \sim q$*

Proposizione 3. *Siano $p, q \in \mathcal{P}$ processi. Allora le seguenti sono valide:*

(1)	$\underline{\mu}.(p + q) \sim \underline{\mu}.p + \underline{\mu}.q$	*(2)*	$\underline{\mu}.0 \sim 0$
(3)	$\underline{\tau}.p$	*(4)*	$\underline{\mu}.\tau.p \sim \mu.p$

Proposizione 4. *Se $p \sim q$ allora le seguenti sono valide:*

1. $\mu.p \sim \mu.q \quad \forall \mu \in Act$
2. $\underline{\mu}.p \sim \underline{\mu}.q \quad \forall \mu \in Act$
3. $p + r \sim q + r \quad \forall r \in \mathcal{P}$
4. $(\nu\, a)p \sim (\nu\, a)q \quad \forall a \in \mathcal{L}$

Sfortunatamente $\sim$ non è una congruenza per la composizione parallele, mostriamo un esempio.

Esempio 7 (Non congruenza per la composizione parallela)**.** *Consideriamo il processo $p = \overline{a}.\overline{a}.0$ e il processo $q = \overline{a}.0 \mid \overline{a}.0$. Chiaramente $p \sim q$. Comunque se consideriamo il contesto $\mathcal{C}[-] = - \mid \underline{a}.\underline{a}.c.0$ noi abbiamo che $\mathcal{C}[p] \nsim \mathcal{C}[q]$ perché q può eseguire c, cioé, $\mathcal{C}[q] \xrightarrow{c} (0 \mid 0) \mid 0$, mentre $\mathcal{C}[p]$ no. La ragione di questa differenza è che i processi $\underline{a}.\underline{a}.c.0$ e $\overline{a}.0 \mid \overline{a}.0$ hanno lo stesso numero di azioni concorrenziali, cosa che non accade con il processo p.*

5.2 Semantica a Step

Multi-CCS può essere dotato di una semantica a step, cioè, una semantia dove ogni transizione è eticettata da un multi-insieme di sequenze che

ogni sottoprocesso concorrente può eseguire contemporaneamente. Questa equivalena è stata originariamente introdotta nelle reti di Petri, qui abbiamo dimostrato che può essere inserito direttamente nei sistemi di transizione.

La semantica operazionale a step per il nostro Multi-CCS è dato dall'lts $(\mathcal{P}, \mathcal{B}, \rightarrow_s)$, dove gli stati sono i processi in $\mathcal{P}, \mathcal{B} = \mathcal{M}_f in(\mathcal{A})$ è il più piccolo insieme di etichette (simboleggiato da M) e $\rightarrow_s \subseteq \mathcal{P} \times \mathcal{B} \times \mathcal{B}$ il più piccola transizione generate dalle regole definite nella Tabella 5.1. Da notare che le regole $(S - pref_1^s)$, $(S - pref_2^s)$, $(S - pref_3^s)$ assumo che le premesse delle transiioni sono sequenziali, cioè composte da una singola sequenza. Da notare anche che la regola $(S - Com^s)$ usa la relazione ausiliaria $MSync$, definita in Tabella 5.2, dove $\oplus$ denota l'unione di multi-insiemi. L'intuiione che c'è dietro la definione della regola $(S - Com^s)$ e $MSync$ è che, ogni volta che due processi paralleli p e q eseguono passi M_1 e M_2, allora possiamo mettere tutte le sequenze insieme - $M_1 \oplus M_2$- e vedere se $MSync(M_1 \oplus M_2, \overline{M})$ è valida. Il risultato $\overline{M}$ può essere $M_1 \oplus M_2$ (quindi non avviene la sincronizzazione), secondo l'assioma $Msync(M, M)$, oppure viene applicata la regola: seleziona due sequenze σ_1 e σ_2 da $M_1 \oplus M_2$, sincronizarli producendo σ, poi ricorsivamente applicare $MSync$ a $M_1 \oplus M_2 \setminus \{\sigma_1, \sigma_2\} \cup \sigma$ per ottenere M'. Questa procedura di sincroniaione può andare avanti fino a trovare tutte le coppie di sincronizazione posibili fino a quando non si ferma a causa dell'assioma $Msync(M, M)$. È interessante osservare che questa semantica operazionale a step non fa uso di congruenza strutturale $\equiv$. Lo stesso effetto della regola (Cong) è qui assicurata dalla relazione $MSync$ che consente la sincronizzazione multipla di sottoprocessi attivi contemporaniamente.

Sincronizazione multi-party

Continuamo la sincronizzazione multi-party risolto con il Multi-CCS nel Paragrafo 2.4.1, con il processo P' = $(\nu\, a, b)(p \,|\, (q \,|\, r))$, dove $p = \underline{a}.b.p'$, $q = \overline{b}.q'$ e $r = \overline{a}.r'$. Scopo di questo paragrafo è mostrare che è possibile, con la semantica a step quindi senza la regola di congruenza, la transizione $P' \xrightarrow{\tau} (\nu\, a, b)(p' \,|\, (q' \,|\, r'))$.

(Pref^s)	$\mu.p \xrightarrow{\{\mu\}}_s p$	(S-Pref$^s_1$)	$\dfrac{p \xrightarrow{\{\sigma\}}_s p'}{\underline{\tau}.p \xrightarrow{\{\sigma\}}_s p'}$	
(S-Pref$^s_2$)	$\dfrac{p \xrightarrow{\tau}_s p'}{\underline{\alpha}.p \xrightarrow{\{\alpha\}}_s p'}$	(S-Pref$_3$)	$\dfrac{p \xrightarrow{\{\sigma\}}_s p' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\{\alpha\sigma\}}_s p'}$	
(Sum$^s_1$)	$\dfrac{p \xrightarrow{M}_s p'}{p + q \xrightarrow{M}_s p'}$	(Sum$^s_2$)	$\dfrac{q \xrightarrow{M}_s q'}{p + q \xrightarrow{M}_s q'}$	
(Par$^s_1$)	$\dfrac{p \xrightarrow{M}_s p'}{p \mid q \xrightarrow{M}_s p' \mid q}$	(S-Com^s)	$\dfrac{p \xrightarrow{M_1}_s p' \qquad q \xrightarrow{M_2}_s q'}{p \mid q \xrightarrow{M}_s p' \mid q'}$	$MSync(M_1 \oplus M_2, M)$
(Par$^s_2$)	$\dfrac{q \xrightarrow{M}_s q'}{p \mid q \xrightarrow{M}_s p \mid q'}$	(S-Res^s)	$\dfrac{p \xrightarrow{\sigma}_s p'}{(\nu\, a)p \xrightarrow{M}_s (\nu\, a)p'}$	$\forall \sigma \in M\ a, \overline{a} \notin n(\sigma)$
(Cons^s)	$\dfrac{p \xrightarrow{M}_s p'}{C \xrightarrow{M}_s p'}$ se: $C \stackrel{def}{=} p$			

Tabella 5.1: Semantica operazionale a Step.

Lemma 3 (Siamo p e $q \in \mathcal{P}$, allora)**.** *:*

1. *se* $p \xrightarrow{\{\sigma\}}_s q$*, allora* $p \xrightarrow{\sigma} q$
2. *se* $p \xrightarrow{\sigma} q$*, allora* $\exists q' \equiv q$ *tale che* $p \xrightarrow{\{\sigma\}}_s q'$

Dimostrazione. La dimostrazione di (1) è per induzione sull'albero di dimostrazione di $p \xrightarrow{\{\sigma\}}_s q$ Nel caso della regola $(S - Com^s)$ la prova effettiva è data dalla regola $MSync$, che dice in quale ordine i paralleli che compongono i sotto-processi devono essere disposti per la congruenza strutturale. La dimostrazione del (2) è data dall'albero di dimostrazione di $p \xrightarrow{\sigma} q$. Noi non siamo in grado di dimostrare il risultato più forte $p \xrightarrow{\{\sigma\}}_s q$ a causa del libero

$$MSync(M, M) \qquad \frac{Sync(\sigma_1, \sigma_2, \sigma) \qquad MSync(M \oplus \{\sigma\}, M')}{MSync(M \oplus \{\sigma_1\sigma_2\}, M')}$$

Tabella 5.2: Relazione di sincronizzazione.

uso della congruenza strutturale; esempio $\mu.(p \,|\, (q \,|\, r)) \xrightarrow{\mu} ((p \,|\, q) \,|\, r)$ (dovuto all'uso della $(Cong)$), mentre $\mu.(p \,|\, (q \,|\, r))$ non può raggiungere $((p \,|\, q) \,|\, r)$ nel sistema di transizione.

□

Quindi, la bisimulazione sul sistema di trasizione a step del Multi-CCS, chiamata step equivalence e denominata con $\sim_{step}$, è più distriminante della ordinaria bisimulazione ($\sim$). Per esempio $(a.0 \,|\, b.0) \sim (a.0 + b.0)$ ma non sono step bisimilar, solo il primo è in grado di eseguire una transizione etichettata $\{a, b\}$

Proposizione 5 (Per ogni processo $p, q \in \mathcal{P}$, se $p \sim_{step} q$ allora $p \sim q$)**.**

Dimostrazione. Al Lemma 4 tutte le transizioni interleaving sono rappresentate all'interno del sistema di transizione a step. Per questo se $p \sim_{step} q$ allora $p \sim q$. □

Teorema 5 (Congruenza)**.** *Se $p \sim_{step} q$ allora valgono le seguenti:*

1. $\mu.p \sim_{step} \mu.q \quad \forall \mu \in Act$
2. $\underline{\mu}.p \sim_{step} \underline{\mu}.q \quad \forall \mu \in Act$
3. $p + r \sim_{step} q + r \quad \forall r \in \mathcal{P}$
4. $p \,|\, r \sim_{step} q \,|\, r \quad \forall r \in \mathcal{P}$
5. $(\nu\, a)p \sim_{step} (\nu\, a)q \quad \forall a \in \mathcal{L}$

Dimostrazione. □

$Sync(\alpha, \overline{\alpha}, \tau)$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma)}{Sync(\alpha\sigma_1, \overline{\alpha}\sigma_2, \sigma)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \tau)}{Sync(\alpha\sigma_1, \sigma_2, \alpha)}$
$\dfrac{Sync(\sigma_1, \sigma_2, \tau)}{Sync(\sigma_1, \alpha\sigma_2, \alpha)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Sync(\alpha\sigma_1, \sigma_2, \alpha\sigma)}$	$\dfrac{Sync(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Sync(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$
$Int(\alpha, \overline{\alpha}, \tau)$	$Int(\alpha, \epsilon, \alpha)$	$Int(\epsilon, \alpha, \alpha)$
$\dfrac{Int(\sigma_1, \sigma_2, \sigma)}{Int(\alpha\sigma_1, \overline{\alpha}\sigma_2, \sigma)}$	$\dfrac{Int(\sigma_1, \sigma_2, \tau)}{Int(\alpha\sigma_1, \sigma_2, \alpha)}$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Int(\alpha\sigma_1, \sigma_2, \alpha\sigma)}$
$\dfrac{Int(\sigma_1, \sigma_2, \tau)}{Int(\sigma_1, \alpha\sigma_2, \alpha)}$	$\dfrac{Int(\sigma_1, \sigma_2, \sigma) \quad \sigma \neq \tau}{Int(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$	

Tabella 5.3: Relazioni di sincronizzazione "$Sync$" e di Interleaving "Int".

Il Teorema 5(4) e la Proposizione 5 garantiscono che per ogni processo $p, q \in \mathcal{P}$ se $p \sim_{step} q$ allora per ogni $r \in \mathcal{P}$, $p \,|\, r \sim q \,|\, r$. È lecito chiedersi se il contrario è valido, cioè se per ogni $r \in \mathcal{P}$, $p \,|\, r \sim q \,|\, r$ possimo concludere che $p \,|\, r \sim_{step} q \,|\, r$? Se questo fosse vero allora la step equivalence sarebbe la più grossolana congruenza contenuta della bisimulazione ($\sim$).La risposta a questa domanda è negativa.

5.3 Congruenza più grossolana

Tanto per un interesse puramente teorico, possiamo chiederci qual è il più grossolana (cioè astratta) congruenza contenuta dalla bisimulazione. Abbiamo osservato sopra che la step bisimilarity, anche se una congruenza, non è la più grossolana possibile.

Si definisce per questo motivo una terza semantica operazionale per il Multi-CCS. La semantica operazionale linear-step è data dall'lts $(\mathcal{P}, \mathcal{A}, \rightarrow_{ls})$, dove $\rightarrow_{ls} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ è il più piccolo relazione di transizione generata dalle regole elencate nella Tabella 5.4. Da notare inoltre che la regola $(S - Com_l s)$ utilizza un ulteriore relazione ausiliaria $Ainit$ definita della Tabella 5.5, che è

(Pref^{ls})	$\mu.p \xrightarrow{\mu}_{ls} p$	(S-Pref$_1^{ls}$)	$\dfrac{p \xrightarrow{\sigma}_{ls} p'}{\underline{\tau}.p \xrightarrow{\sigma}_{ls} p'}$	
(S-Pref$_2^{ls}$)	$\dfrac{p \xrightarrow{\tau}_{ls} p'}{\underline{\alpha}.p \xrightarrow{\alpha}_{ls} p'}$	(S-Pref$_3^{ls}$)	$\dfrac{p \xrightarrow{\sigma}_{ls} p' \qquad \sigma \neq \tau}{\underline{\alpha}.p \xrightarrow{\alpha\sigma}_{ls} p'}$	
(Sum$_1^{ls}$)	$\dfrac{p \xrightarrow{\sigma}_{ls} p'}{p + q \xrightarrow{\sigma}_{ls} p'}$	(Sum$_2^{ls}$)	$\dfrac{q \xrightarrow{\sigma}_{ls} q'}{p + q \xrightarrow{\sigma}_{ls} q'}$	
(Par$_1^{ls}$)	$\dfrac{p \xrightarrow{\sigma}_{ls} p'}{p \mid q \xrightarrow{\sigma}_{ls} p' \mid q}$	(S-Com^{ls})	$\dfrac{p \xrightarrow{\sigma_1}_{ls} p' \qquad q \xrightarrow{\sigma_2}_{ls} q'}{p \mid q \xrightarrow{\sigma}_{ls} p' \mid q'}$	$AInt(\sigma_1, \sigma_2, \sigma)$
(Par$_2^{ls}$)	$\dfrac{q \xrightarrow{\sigma}_{ls} q'}{p \mid q \xrightarrow{\sigma}_{ls} p \mid q'}$	(S-Res^{ls})	$\dfrac{p \xrightarrow{\sigma}_{ls} p'}{(\nu\, a)p \xrightarrow{\sigma}_{ls} (\nu\, a)p'}$	$\forall \sigma \in M\ a, \overline{a} \notin n(\sigma)$
(S-Cons^{ls})	$\dfrac{p \xrightarrow{\sigma}_{ls} p'}{C \xrightarrow{\sigma}_{ls} p'}$ se: $C \stackrel{def}{=} p$			

Tabella 5.4: Semantica operazionale per Linear Step.

un'estenzione di $Init$, definito nella Tabella 5.3. Si noti che se $AInt(\sigma_1, \sigma_2, \sigma)$ vale, allora σ è o τ oppure è una sequenza commposta solo da azioni osservabili. Si osservi inoltre che l'unica vera differena tra la semantica interleaving e quella linear step è che $AInt(\sigma_1, \sigma_2, \sigma)$ consente l'avanamento singolo di σ_1 o σ_2, mentre $Sync(\sigma_1, \sigma_2, \sigma)$ vale solo se almeno una sincronizzazione avviene.

Lemma 4 (Siamo p e $q \in \mathcal{P}$, allora)**.** *:*

1. *se* $p \xrightarrow{\{\sigma\}}_{ls} q$*, allora* $p \xrightarrow{\sigma} q$
2. *se* $p \xrightarrow{\sigma} q$*, allora* $\exists q' \equiv q$ *tale che* $p \xrightarrow{\sigma}_{ls} q'$

Lemma 5 (Siano p e $q \in \mathcal{P}$. Se $p \equiv q$ allora $p \sim_{ls} q$)**.**

$$AInt(\tau,\sigma,\sigma) \qquad AInt(\sigma,\tau,\sigma) \qquad \frac{\sigma_1 \neq \tau \qquad \sigma_2 \neq \tau \qquad Int(\sigma_1\sigma_2,\sigma)}{AInt(\sigma_1\sigma_2,\sigma)}$$

$$Int(\alpha,\overline{\alpha},\tau) \qquad Int(\alpha,\epsilon,\alpha) \qquad Int(\epsilon,\alpha,\alpha)$$

$$\frac{Int(\sigma_1,\sigma_2,\sigma)}{Int(\alpha\sigma_1,\overline{\alpha}\sigma_2,\sigma)} \qquad \frac{Int(\sigma_1,\sigma_2,\tau)}{Int(\alpha\sigma_1,\sigma_2,\alpha)} \qquad \frac{Int(\sigma_1,\sigma_2,\sigma) \qquad \sigma \neq \tau}{Int(\alpha\sigma_1,\sigma_2,\alpha\sigma)}$$

$$\frac{Int(\sigma_1,\sigma_2,\tau)}{Int(\sigma_1,\alpha\sigma_2,\alpha)} \qquad \frac{Int(\sigma_1,\sigma_2,\sigma) \qquad \sigma \neq \tau}{Int(\sigma_1,\alpha\sigma_2,\alpha\sigma)}$$

Tabella 5.5: Relazione di sincronizzazione.

L'ordinaria bisimulazione sulla semantica operazionale linear-step è chiamata equivalenza linear step e denotata con $\sim_{ls}$. Proviamo che $\sim_{ls}$ è una congruenza per il Multi-CCS.

Teorema 6 (Congruenza)**.** *Se $p \sim_{ls} q$ allora valgono le seguenti:*

1. $\mu.p \sim_{ls} \mu.q \quad \forall \mu \in Act$
2. $\underline{\mu}.p \sim_{ls} \underline{\mu}.q \quad \forall \mu \in Act$
3. $p + r \sim_{ls} q + r \quad \forall r \in \mathcal{P}$
4. $p \,|\, r \sim_{ls} q \,|\, r \quad \forall r \in \mathcal{P}$
5. $(\nu\, a)p \sim_{ls} (\nu\, a)q \quad \forall a \in \mathcal{L}$

Teorema 7 (Congruenza più grossolana)**.** *Assmiamo che $fn(p) \cup fn(q) \neq \mathcal{L} \cup \overline{\mathcal{L}}$. Allora, $p \sim_{ls} q$ se e solo se $\forall r \in \mathcal{P}$, $p \,|\, r \sim q \,|\, r$.*

Capitolo 6

Interprete per Multi-CCS-ls: Multi-muCCS-ls

A questo punto possiamo analizzare l'interprete del Multi-CCS-ls, appunto Multi-muCCS-ls, la sua semantica, quindi la facile ed immediata dimostrazione della sua correttezza e completezza.

Il progetto nasce dalla modifica della semantica dell'interprete Multi-muCCS, quindi del solo file PROLOG, questo per dire che tutto il resto del tool resta invariato. Per motivi di sintesi e vista l'ormai acquisita padronanza nella manipolazione e trasformazione in PROLOG di regole SOS, in questo capitolo della dissertazione discuteremo della sola semantica espressa dal Multi-muCCS-ls, quindi le regole implementate verranno scritte in linguaggio matematico e non più in PROLOG.

6.1 Le regole SOS nel Multi-muCCS-ls

Introduciamo in questo paragrafo le regole realmente implementate in PROLOG (che chiameremo insieme delle regole D) per la realizaione dell'interprete Multi-muCCS-ls, dimostrando successivamente l'equivalenza con le regole originali.

(D-Pref^{ls})	$\mu.p \overset{\mu}{\rightsquigarrow}_{ls} p$	(D-S-Pref$^{ls}_1$)	$\dfrac{p \overset{\sigma}{\rightsquigarrow}_{ls} p'}{\underline{\tau}.p \overset{\sigma}{\rightsquigarrow}_{ls} p'}$	$\sigma \neq []$
(S-Pref$^{ls}_2$)	$\dfrac{p \overset{\tau}{\rightsquigarrow}_{ls} p'}{\underline{\alpha}.p \overset{\alpha}{\rightsquigarrow}_{ls} p'}$	(D-S-Pref$^{ls}_3$)	$\dfrac{p \overset{\sigma}{\rightsquigarrow}_{ls} p' \qquad \sigma \neq \tau}{\underline{\alpha}.p \overset{\alpha\sigma}{\rightsquigarrow}_{ls} p'}$	$\sigma \neq []$
(D-Sum$^{ls}_1$)	$\dfrac{p \overset{\sigma}{\rightsquigarrow}_{ls} p'}{p + q \overset{\sigma}{\rightsquigarrow}_{ls} p'}$	(D-Sum$^{ls}_2$)	$\dfrac{q \overset{\sigma}{\rightsquigarrow}_{ls} q'}{p + q \overset{\sigma}{\rightsquigarrow}_{ls} q'}$	
(D-Par$^{ls}_1$)	$\dfrac{p \overset{\sigma}{\rightsquigarrow}_{ls} p'}{p \mid q \overset{\sigma}{\rightsquigarrow}_{ls} p' \mid q}$	(D-S-Com^{ls})	$\dfrac{p \overset{\sigma_1}{\rightsquigarrow}_{ls} p' \qquad q \overset{\sigma_2}{\rightsquigarrow}_{ls} q'}{p \mid q \overset{\sigma}{\rightsquigarrow}_{ls} p' \mid q'}$	$aint_D(\sigma_1, \sigma_2, \sigma)$
(D-Par$^{ls}_2$)	$\dfrac{q \overset{\sigma}{\rightsquigarrow}_{ls} q'}{p \mid q \overset{\sigma}{\rightsquigarrow}_{ls} p \mid q'}$	(D-S-Res^{ls})	$\dfrac{p \overset{\sigma}{\rightsquigarrow}_{ls} p'}{(\nu\, A)p \overset{\sigma}{\rightsquigarrow}_{ls} (\nu\, A)p'}$	$not\_in\_L(\sigma, A)$
(D-S-Cons^{ls})	$\dfrac{p \overset{\sigma}{\rightarrow}_{ls} p'}{proc(K) \overset{\sigma}{\rightarrow}_{ls} p'}$	se:	proc_def(K, P)	

Tabella 6.1: Semantica operazionale per Linear Step.

6.1.1 Equivalenza delle regole in PROLOG

Come abbiamo già detto dell'introduzione del capitolo, la "traduzione" delle regole SOS in PROLOG risulta, a questo punto della lettura, abbastanza banale; esiste una corrispondenza uno a uno tra le regole di definizione e quelle utilizzate dall'interprete. La semantica operazionale in discusione infatti non ha particolari regole, come la regola ($Cong$) del Multi-muCCS, di difficile traduzione. Teniamo sottocchio le regole di semantica operazionale riportate in Tabella 5.4, in Tabella 5.5, le regole implementate riportate rispettivamente nelle in Tabella 6.1, Tabella 6.2 e Tabella 6.3.

Possiamo dimotrare quindi che il la semantica operazionale raccolte dell'insieme D equivale alle regole di semantica operazionale originali dell'Multi-CCS-ls

Iniziamo con la spiegazione e dimostrazione delle regole a margine delle

$$aint_D(\tau,\sigma,\sigma) \qquad aint_D(\sigma,\tau,\sigma) \qquad \frac{\sigma_1 \neq \tau \qquad \sigma_2 \neq \tau \qquad int_D(\sigma_1\sigma_2,\sigma)}{aint_D(\sigma_1\sigma_2,\sigma)}$$

Tabella 6.2: Relazione di sincronizzazione.

$$int_D(\alpha,\overline{\alpha},\tau) \qquad int_D(\alpha,\epsilon,\alpha) \qquad int_D(\epsilon,\alpha,\alpha)$$

$$\frac{int_D(\sigma_1,\sigma_2,\sigma)}{int_D(\alpha\sigma_1,\overline{\alpha}\sigma_2,\sigma)} \qquad \frac{int_D(\sigma_1,\sigma_2,\tau)}{int_D(\alpha\sigma_1,\sigma_2,\alpha)} \qquad \frac{int_D(\sigma_1,\sigma_2,\sigma) \qquad \sigma \neq \tau}{int_D(\alpha\sigma_1,\sigma_2,\alpha\sigma)}$$

$$\frac{int_D(\sigma_1,\sigma_2,\tau)}{int_D(\sigma_1,\alpha\sigma_2,\alpha)} \qquad \frac{int_D(\sigma_1,\sigma_2,\sigma) \qquad \sigma \neq \tau}{int_D(\sigma_1,\alpha\sigma_2,\alpha\sigma)}$$

Tabella 6.3: Semantica operazionale per Linear Step.

deduzioni dell'insieme D.

Nella regola (D-S-pref$_1$) è presente la condizione $\sigma \neq []$. Questo per imporre all'interprete prolog che non sono accettate soluzioni che vedono la variabile σ come una stringa vuota, soluzioni invece accettate in alcune regole della relazione int_D, in queste la stringa vuota è defineta dalla lettera greca ϵ. Ancora, nella regola (D-S-Cons) è presente come condizione la relazione $proc_def(K,P)$, questa funge da dichiarazione di costante di processo. Quindi porre K definita e P veriabile prolog equivale alla condizione: se $K \stackrel{def}{=} P$. Arriviamo quindi a dimostrare che le relazioni $aint_D$ equivalgono alle relazioni *AInt*.

Lemma 6 (Le relazioni *Int* e int_D sono equivalenti)**.**

Dimostrazione. La dimostrazione è per induzione sulla struttura delle relazioni:

base: $Int(\alpha,\overline{\alpha},\tau)$ corrisponde a $int_D(\alpha,\overline{\alpha},\tau)$

$Int(\alpha,\epsilon,\alpha)$ corrisponde a $int_D(\alpha,\epsilon,\alpha)$

$Int(\epsilon,\alpha,\alpha)$ corrisponde a $int_D(\epsilon,\alpha,\alpha)$

induzione: il passo induttivo a questo punto è banale, presentiamo un solo caso.

$\dfrac{Int(\sigma_1, \sigma_2, \sigma) \qquad \sigma \neq \tau}{Int(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$ corrisponde a $\dfrac{int_D(\sigma_1, \sigma_2, \sigma) \qquad \sigma \neq \tau}{Int(\sigma_1, \alpha\sigma_2, \alpha\sigma)}$

□

Lemma 7 (Le relazioni $AInt$ e $aint_D$ sono equivalenti)**.**

Dimostrazione. La dimostrazione è immediata, basta notare infatti la corrispondenza uno a uno tra regole delle due relazioni.

Passo base:

$aint_D(\tau, \sigma, \sigma)$ corrisponde a $AInt(\tau, \sigma, \sigma)$

Passo induttivo:

$aint_D(\sigma, \tau, \sigma)$ corrisponde a $AInt(\sigma, \tau, \sigma)$

$$\frac{\sigma_1 \neq \tau \qquad \sigma_2 \neq \tau \qquad int_D(\sigma_1\sigma_2, \sigma)}{aint_D(\sigma_1\sigma_2, \sigma)} \text{ corrisponde a } \frac{\sigma_1 \neq \tau \qquad \sigma_2 \neq \tau \qquad Int(\sigma_1\sigma_2, \sigma)}{AInt(\sigma_1\sigma_2, \sigma)}$$

□

Teorema 8 (Se $p \xrightarrow{\sigma}_{ls} q \iff p \overset{\sigma}{\rightsquigarrow}_{ls} q$)**.**

Dimostrazione. ($\Rightarrow$) Per induzione sulla dimostrazione di $p \xrightarrow{\sigma}_{ls} q$. Il caso base è quando $p = \mu.q$, $\sigma = \mu$ e viene usato l'assioma (Pref^{ls}). In questo caso $p \overset{\mu}{\rightsquigarrow}_{ls} q$ deriva dal assioma (D-Pref^{ls}). I caso induttivi invece sono triviali; presentiamo la dimostrazione della regola (S-Com^{ls}), dove $p = p_1 \,|\, p_2$ e $p_1 \,|\, p_2 \xrightarrow{\sigma}_{ls} p_1' \,|\, p_2'$ è permesso da $p_1 \xrightarrow{\sigma_1}_{ls} p_1'$ e da $p_2 \xrightarrow{\sigma_2}_{ls} p_2'$ e dalla regola a margine $AInt(\sigma_1, \sigma_2, \sigma)$. Per induzione abbiamo che $p_1 \overset{\sigma_1}{\rightsquigarrow}_{ls} p_1'$ e da $p_2 \overset{\sigma_2}{\rightsquigarrow}_{ls} p_2'$ quindi $p_1 \,|\, p_2 \overset{\sigma}{\rightsquigarrow}_{ls} p_1' \,|\, p_2'$.

($\Leftarrow$) Allo stesso modo, per induzione sull'albero di derivazione di $p \overset{\sigma}{\rightsquigarrow}_{ls} q$

□

Conclusioni

Abbiamo studiato il problema di trovare una semantica in grado di fornire l'atomicità tra le sue caratteristiche. A questo obbiettivo abbiamo scelto un operatore semplice (strong_ prefix) e un'interazione in grado di disciplinare i comportamenti atomici (relazione Sync), ottenendo appunto il Multi-CCS. Il modello di interleaving identifica tutte le effettive (cioè fisiche) transizioni atomiche di un processo. Tuttavia abbiamo visto che l'interleaving bisimulation non è una congruenza rispetto all'operatore di parallelo. Pertanto è risultato necessario arricchire il modello con cinque transizioni supplementari, gli assiomi di congruenza strutturale per il Multi-CCS. Per rendere più facile l'implementazione dell'interprete del linguaggio abbiamo deciso di modificare, rispetto all'originale, la definizione della sintassi, modificare la definizione di semantica operazionale permettendoci di eliminare quindi l'assioma per la costante che rendeva non sempre vero il lemma di navigazione della classe CA; questa scelta è risultata la migliore perché rende comunque la bisimilarità una congruenza e preserva la bisimilarità tra i processi appartenenti ai due linguaggi.

Abbiamo ottenuto così la non più grossolana congruenza possibile, per questo scopo abbiamo definito e implementato, tanto per un interesse puramente teorico, la semantica linear-step , che risulta essere la congruenza più grossolana. Il lavoro di implementazione e di dimostrazione della correttezza e della completezza di questa semantica è stato decisamente più semplice, questo perché la semantica elimina i cinque assiomi presenti nella semantica precedente.

Quindi come sviluppo futuro abbiamo quello di trovare un modo più efficace e furbo per includere, all'interno dell'implementazione del Multi-CCS, l'assioma di congruenza strutturale "bypassato" in questa versione.

Appendice A

sostituzione sintattica e α-conversione

A.1 sostituzione sintattica senza cattura

Prima di dare la definizione di sostituzione sintattica e α-conversione, definiamo alcune alcune notazioni ausiliarie indispensabili allo scopo [Bar85, Mil89]:

Def 14. *L'insieme dei nomi liberi di un processo p, denotato con $fn(p)$, è definito come l'insieme $F(p, \varnothing)$, dove $F(p, I)$, con I insieme delle costanti, è definita come segue:*

$$
\begin{array}{rcl}
F : Proc \times Proc \to \mathcal{P}(Act) & & \\
F(0, I) & = & \varnothing \\
F(\mu.p, I) & = & F(p, I) \cup \{\mu\} \\
F(\underline{\mu}.p, I) & = & F(p, I) \cup \{\mu\} \\
F(p + q, I) & = & F(p, I) \cup F(q, I) \\
F(p \,|\, q, I) & = & F(p, I) \cup F(q, I) \\
F((\nu\, a)p, I) & = & F(p, I) \setminus \{a, \overline{a}\} \\
F(A, I) & = & \begin{cases} F(q, I \cup A) & se \quad A \stackrel{def}{=} q \quad e \quad A \notin I \\ \varnothing & se \quad A \in I \end{cases}
\end{array}
$$

□

Def 15. *I nomi legati di un processo p, denotato con bn(p), sono definiti come l'insieme $B(p,0)$, dove $B(p.I)$, con I insieme delle costanti, è definita come segue:*

$$
\begin{array}{l}
B : Proc \times \mathcal{P}(Proc) \to \mathcal{P}(Act) \\
\begin{array}{rcl}
B(0,I) & = & \varnothing \\
B(\mu.p,I) & = & B(p,I) \\
B(\underline{\mu}.p,I) & = & B(p,I) \\
B(p\,+\,q,I) & = & B(p,I) \cup B(q,I) \\
B(p\,|\,q,I) & = & B(p,I) \cup B(q,I) \\
B((\nu\,a)p,I) & = & B(p,I) \cup \{a,\overline{a}\} \\
B(A,I) & = & \begin{cases} B(q,I \cup A) & se \quad A \stackrel{def}{=} q \quad e \quad A \notin I \\ \varnothing & se \quad A \in I \end{cases}
\end{array}
\end{array}
$$

□

Def 16. *I nomi di un processo p, denotato da n(p) sono definiti come segue:*

$n : Proc \to \mathcal{P}(Act)$

$n(p) = F(p,0) \cup B(p.0)$

□

Def 17. *La sostituzione sintattica $p\{b/a\}$ del nome b al posto del nome a nel processo p ($\{\cdot\} : Proc \to Act \times Act \to Proc$), è definita come segue:*

$$
\begin{array}{rcccl}
0\{b/a\} & = & 0 & & \\
(a.p)\{b/a\} & = & b.(p\{b/a\}) & & \\
(\overline{a}.p)\{b/a\} & = & \overline{b}.(p\{b/a\}) & & \\
(\underline{a}.p)\{b/a\} & = & \underline{b}.(p\{b/a\}) & & \\
(\overline{\underline{a}}.p)\{b/a\} & = & \overline{\underline{b}}.(p\{b/a\}) & & \\
(\mu.p)\{b/a\} & = & \mu.(p\{b/a\}) & se & \mu \neq a \\
(\underline{\mu}.p)\{b/a\} & = & \underline{\mu}.(p\{b/a\}) & se & \mu \neq a \\
p + p'\{b/a\} & = & p\{b/a\} + p'\{b/a\} & & \\
p \,|\, p'\{b/a\} & = & p\{b/a\} \,|\, p'\{b/a\} & & \\
((\nu\, a)q)\{b/a\} & = & (\nu\, a)q & & \\
((\nu\, b)q)\{b/a\} & = & (\nu\, c)((q\{c/b\})\{b/a\}) & con & c \text{ fresca} \\
((\nu\, c)q)\{b/a\} & = & (\nu\, c)(q\{b/a\}) & se & c \neq a, b \\
A\{b/a\} & = & A_{\{b/a\}} & dove & A_{\{b/a\}} \stackrel{def}{=} q\{b/a\} \quad se \quad A \stackrel{def}{=} q
\end{array}
$$

Nota 5. *Al fine di semplificare la notazione, date due sostituzioni f e f' identifichiamo $A_{ff'}$ con $A_{f \circ f'}$ ma che, per esempio, $A_{\{a'/a\}\{a''/a'\}} = AA_{\{a''/a\}}$*

□

A.2 α-conversione

A questo punto data la definizione di sostituzione sintattica la regola di α-conversione viene data sotto forma di proposizione:

Proposizione 6. *per ogni processo p si definisce ottiene α-conversione nel seguente caso:*

$$(\nu\, a)p \sim (\nu\, b)(p\{b/a\}) \quad se \quad b \notin fn(p)$$

□

Appendice B

La programmazione logica

In questo capitolo, tratto da [GM06], analizziamo un paradigma che, insieme a quello funzionale, permette la programmazione dichiarativa: il paradigma logico che include sia linguaggi teorici sia linguaggi effettivamente implementati ed usati, dei quali il più noto è sicuramente PROLOG. Anche se vi sono differenze abbastanza rilevanti di ordine pragmatico e, per certi versi, teorico, fra i vari linguaggi logici, questi condividono l'idea di intendere la computazione come deduzione logica.

Questo capitolo vuole fornire le basi del paragigma logico in modo da poter comprendere l'implementazione delle regole SOS e la conseguente dimostrazione di equivalenza con le regole originali del Multi-CCS.

B.1 Deduzione come computazione

Un noto slogan, dovuto a R. Kowalski, così sintetizza la nozione alla base dell'attività di programmazione: Algoritmo = Logica + Controllo. Secondo questa "equazione" la specifica di un algoritmo, e quindi la sua formulazione in termini di un linguaggio di programmazione, può essere separata in due parti. Da un lato si specifica la logica della soluzione, ossia si definisce "cosa" debba essere fatto. Dall'altro invece si specificano gli aspetti relativi al con-

trollo, e quindi si chiarisce "come" si deve arrivare alla soluzione desiderata[1]. La programmazione logica separa nettamante i due aspetti, al programmatore è richiesta solo, almeno in linea di principio, la specifica della porte logica. Tutto quanto riguarda il controllo è demandato alla macchina astratta: usando un meccanismo computazionale basato su una particolare regola di deduzione (la risoluzione), essa ricerca nello spazio delle possibili soluzioni quella specificata dalla "logica", definendo in questo modo la sequenza di operazioni necessarie al risultato finale.

B.2 Sintassi Prolog

I programmi logici sono insiemi di formule logiche di una particolare forma. Inizieremo quindi con alcune nozioni di base necessarie per definire la sintassi.

La logica della quale ci occupiamo è quella del prim'ordine, detta anche calcolo dei predicati.

B.2.1 Il linguaggio della logica del prim'ordine

Innanzitutto, come per ogni altro sistema formale, per poter parlare del calcolo dei predicati (e quindi dei programmi logici) dobbiamo definire il linguaggio. Un linguaggio del prim'ordine consiste di tre componenti:

1. un alfabeto;

2. i termini su tale alfabeto;

3. le formule ben formate definite su tale alfabeto.

[1] Il programmatore che utilizza un tradizionale linguaggio imperativo, deve tener conto di entrambe queste componenti.

Alfabeto

L'alfabeto, al solito, è un insieme[2] di simboli. In questo caso consideriamo tale insieme partizionato in due sottoinsiemi disgiunti: l'insieme dei simboli logici, comuni a tutti i linguaggi del prim'ordine, e l'insieme dei simboli non logici, specifici di un qualche dominio di interesse.

L'insieme dei simboli logici contiene elementi come connettivi logici, costanti, quantificatori, simboli di interruzione, insieme finito (numerabile) di variabili.

I simboli non logici sono definiti da una segnatura con predicati $< \Sigma, \Pi >$: si tratta di una coppia nella quale Σ è la segnatura delle funzioni. Il secondo elemento Π è la segnatura dei predicati. Le funzioni con arietà 0 sono dette costanti. Assumiamo che gli insiemi Σ e Π abbiano intersezione vuota e siano anche disgiunti dagli insiemi logici.

Termini

La nozione di termini, fondamentale nella logica matematica. Nel caso più semplice un termine è ottenuto applicando un simbolo di funzione e variabili e costanti in modo tale da ispettare l'arietà.

Def 18 (Termini)**.** *I termini sulla segnatura Σ (e sull'insieme di veriabili V) sono definiti induttivamente come segue:*

- *una variabile (in V) è un termine;*
- *se f (in Σ) è un simbolo di funzione di arietà n e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine.*

Formule

Le formule ben formate (fbf) del linguaggio ci permettono di esprimere proprietà dei termini, che poi dal punto di vista semantico, sono proprietà di un particolare dominio di interesse.

[2]Finito o al più numerabile.

Def 19 (Formule)**.** *Le formule (ben formate) del linguaggio sulla segnatura con termini* (Σ, Π) *sono definiti induttivamente come segue:*

- *se* $t_1, \ldots, t_n$ *sono termini sulla segnatura* Σ *e* $p \in \Pi$ *è un simbolo di predicato di arietà* n*, allora* $p(t_1, \ldots, t_n)$ *è una formula;*
- *true e* $false$ *sono formule;*
- *se* F *e* G *sono formule allora l'applicazione dei connettivi logici e dei quantificatori alle formule sono ancora formule.*

B.2.2 I programmi logici

Una formula della logica del prim'ordine pu avere una struttura molto complesa, che influisce sulla difficoltà di determinare una dimostraziona per essa. Nell'ambito della dimostrazione automatica di teoremi e, poi, nel contesto della programmazione logica, sono state identificate particolari classi di formule, dette clausole, che si prestano ad essere manipolate più efficientemente, in specie usando una particolare regola di astrazione detta risoluzione. A noi interessano, in particolare, delle versioni ristrette della nozione di clausola (le clausole di Horn Vedi 77) e della regola di risoluzione (la risoluzione SLD) per le quali il procedimento di ricerca di una dimostrazione non solo è praticamente semplice, ma permette di calcolare esplicitamente i valori delle variabili necessari alla dimostrazione. Questi valori possono essere considerati come il risultato della computazione, dando luogo ad un modello di calcolo interamente basato sulla deduzione logica. Vedremo meglio tale modello più avanti, per ora ci concentriamo sugli sugli aspetti sintattici.

Def 20 (Programma logico)**.** *Siano* $H, A_1, \ldots, A_n$ *formule atomiche. Una clausola definita (in breve "clausola") è una formula della forma*

$$H :- A_1, \ldots A_n.$$

Se $n = 0$ *la clausola è detta unitaria, o fatto, ed il simbolo* $:-$ *è omesso (ma non il punto terminale). Un programma logico è un insieme di clausole,*

mentre un programma PROLOG puro è una sequenza di clausole[3]*. Una query (o goal) è una sequenza di atomi* $A_1, \ldots, A_n$.

Clausole di Horn[WC84, AA97]

Una procedura Prolog così come le regole operazionali sono del tipo:

$$conclusione \leftarrow premesse$$

quindi *conclusione* vera se sono vere le *premesse*.
Una clausola di Horn è una disgiunzione di letterali con al più un letterale positivo, ovvero una clausola del tipo

$$\neg x_1 \vee \neg x_2 \vee \ldots \neg x_n \vee y$$

applicando le leggi di De morgan:

$$\neg(x_1 \wedge x_2 \wedge \ldots \wedge x_n) \vee y$$

quindi utilizzando l'equivalenza logica $\neg x \vee y \equiv x \rightarrow y$, ricaviamo:

$$(x_1 \wedge x_2 \wedge \ldots \wedge x_n) \rightarrow y$$

che si traduce in Prolog:

$$y \leftarrow (x_1 \wedge x_2 \wedge \ldots \wedge x_n)$$

B.3 Teoria dell'unificazione

Il meccanismo fondamentale di computazione della programmazione logica è la soluzione di equazioni fra termini, realizzata mediante il processo di unificazione. In tale processo vengono calcolate delle sostituzioni che permettoni di legare (o istanziare) delle variabili a dei termini. La composizione delle varie sostituzioni ottenute nel corso della computazione fornisce il risultato del calcolo.

[3] In PROLOG l'ordine delle clausole ha importanza.

Analizziamo alcuni aspetti riguardanti l'unificazione.

B.3.1 La variabile logica

La variabile Logica costituisce una un'incognita che può assumere i valori di un insieme prefissato, quello dei termini definiti sull'alfabeto dato. Questo unitamente all'uso che della variabile logica viene fatto nel paradigma logico, fa sì che vi si siamo tre importanti differenze fra questa nozione e la variabile modificabile dei linguaggi imperativi:

1. la variabile logica può essere legata una sola volta, nel senso che se una variabile è legata ad un termine questo legame non può essere distrutto.

2. Il valore di una variabile logica può essere definito parzialmente (o indefinito) per essere poi ulteriormente specificato in seguito. Questo perché un legame ad una variabile può contenere, a sua volta, altre variabili logiche.

3. Una terza importante differenza riguarda la natura bidirezionale dei legami nel caso delle variabili logiche. Se X è legata al termine $f(Y)$ e successivamente proviamo a legare X al termine $f(a)$, l'effetto che produciamo è quello di legare X con a.

B.3.2 Sostituzione

Il legame tra variabili e termini è realizzato mediante la nozione di sostituzione che, come dice il nome stesso, permette di "sostituire" un termine al posto di una variabile. Una sostituzione, indicata con una lettera greca, è definita come segue:

Def 21 (Sostituzione)**.** *Una sostituzione è una funzione da variabili a termini tale che il numero di variabili che non sono mappate in se stesse è finito. Indichiamo una sostituzione ϑ usando la notazione*

$$\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}$$

dove $X_1, \ldots, X_n$ sono variabili diverse, $t_1, \ldots, t_n$ sono termini e dove assumiamo che t_i sia diverso da X_i per $i = 1, \ldots, n$

Un particolare tipo di sostituzioni è costituito da quelle che semplicemente ridenominano le variabili.

Def 22 (Ridenominazione)**.** *Una sostituzione ρ è una ridenominazione se esiste la sua sostituzione inversa ρ^{-1} tale che $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$*

B.3.3 L'unificatore più generale

Il meccanismo di computazione di base del paradigma logico è la valutazione di equazioni della forma $s = t$, dove s e t sono termini e "=" è un simbolo di predicato interpretato come uguaglianza sintattica sull'insieme di tutti i termini ground[4], insieme detto anche universo di Hebrand. Se in un programma logico scriviamo $X = a$ intendiamo dire che la variabile X deve essere legata alla costante a. La soluzione $\{X = a\}$ costituisce dunque una soluzione per tale equazione dato che, applicando questa sostituzione all'equazione, otteniamo $a = a$ che è un equazione evidentemente soddisfatta. Detta in terminio più formali, la sostituzione ϑ unifica i due termini dell'equazione ed è pertanto detta unificatore. Abbiamo detto "una soluzione" perché vi sono molte (infinite) sostituzioni che sono unificatori di una equazione. esiste però un'unica soluzione più generale possibile chiamato m.g.u. (most general unifier).

Prima di passare alla definizione di m.g.u., si noti un ultimo particolare importante: il processo di soluzione di un equazione, e cioè di unificazione, può creare dei legami di natura bidirezionale, ossia non è specificata una direzione nella quale si devono realizzare le associazioni. Ad esampio, una soluzione dell'equazione $f(X, a) = f(b, Y)$ è data dalla sostituzione $\{X/b, Y/a\}$ dove si lega una variabile a sinistra e una variabile a destra del simbolo $=$.

[4] insieme dei termini che non contengono variabili

Quest'aspetto è importante perché permette di realizzare un meccanismo di passaggio dei parametri bidirezionale e caratteristica unica del paradigma logico, di usare lo stesso programma in molti modi differenti, facendo diventare gli argomenti di input argomenti di output e viceversa, senza alcuna modifica del programma stesso.

Def 23 (M.g.u.)**.** *dato un insieme di equazioni* $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ *dove* $s_1, \ldots, s_n$ *e* $t_1, \ldots, t_n$ *sono termini, la sostituzione* ϑ *è un unificatore per* E *se le sequenze* $(s_1, \ldots, s_n)\vartheta$ *e* $(t_1, \ldots, t_n)\vartheta$ *sono sintatticamente identiche. Un unificatore di* E *è detto unificatore più generale (m.g.u.) se per ogni altro unificatore* σ *di* E *vale che* σ *è uguale a* $\vartheta\tau$ *per un'opportuna sostituzione* τ*.*

B.4 Il modello computazionale

Volendo sintetizzare l'idea di "computazione come deduzione", idea su cui si basa la computazione del paradigma logico, possiamo individuare le seguenti differenze principali rispetto agli altri paradigmi.

1. Gli unici valori possibili, almeno nel modello puro, sono i termini su una data segnatura.

2. I programmi possono avere una lettura dichiarativa, iteramente logica, o una lettura procedurale di tipo più operazionale.

3. La computazione avviene istanziando le variabili che appaiono nei termini (e quindi nei goal) con altri termini, usando il meccanismo di unificazione.

4. Il controllo, interamente gestito dalla macchina astratta è basato sul meccanismo di backtracking automatico.

Illustriamo brevemente questi quattro punti.

B.4.1 L'universo di Herbrand

L'insieme di tutti i possibili termini su una data segnatura è detto universo di Herbrand e costituisce il dominio sul quale viene effettuata la computazione di un programma logico. Facciamo notare alcune particolarità.

- Ralativamente ai simboli non logici l'alfabeto sul quale sono definiti i programmi non è prefissato ma può variare. Spesso è determinato dai simboli che compaiono nel particolare programma in questione.

- Come conseguenza logica del punto precedente, nessun significato predefinito è associato ai simboli (non logici) dell'alfabeto. Fanno eccezioni alcuni predicati detti "build-in" presenti il PROLOG.

- Come ulteriore conseguenza, nessun sistema di tipi è presente nei linguaggi logici. L'unico tipo presente è quello dei termini con i quali possiamo rappresentare espressioni aritmetiche, liste, ecc.

B.4.2 Interpretazione dichiarativa e procedurale

Come abbiamo già accennato, una clausola, e quindi un programma logico, può avere due interpretazioni diverse: una *dichiarativa* ed una *procedurale*.

Dal punto di vista *dichiarativo*, una clausola $H :- A_1, \ldots, A_n.$ è una formula che, sostanzialmente, esprime che se $A_1, \ldots, A_n$ sono veri allora è vero anche H. Una query (o goal) è anch'essa una formula per la quale vogliamo dimostrare che, se opportunamente istanziata, è una conseguenza logica del programma, ossia vale in tutte le interpretazioni nelle quali vale il programma.

L'interpretazione *procedurale*, invece, permette di leggere una clausola $H :- A_1, \ldots, A_n.$ come segue: per dimostrare H devi prima calcolare $A_1, \ldots, A_n$. Un programma logico è dunque un insieme di dichiarazioni e un goal non è altro che l'equivalente del "main" di un programma imperativo, dato che contiene tutte le chiamate di procedura che si vogliono valutare.

Precisi teoremi di corrispondenza permettono di riconciliare la visione dichiarativa e quella procedurale, dimostrando che i due approcci sono equivalenti.

B.4.3 La chiamata di procedura

Consideriamo per il momento una definizione semplificata di clausola, dove assumiamo che nella testa tutti gli argomenti del predicato siano variabili distinte. Una generica clausola di questo tipo ha dunque la forma $p(X_1, \ldots, X_n.) : -A_1, \ldots, A_m.$, questa può essere vista come la dichiarazione della procedura p con n parametri formali $X_1, \ldots, X_n$. Un atomo $p(t_1, \ldots, t_n.)$ può essere visto come la chiamata della procedura p con gli n parametri attuali $t_1, \ldots, t_n$. Nella definizione di p, dunque, il corpo è costituito dell'invocazione delle m procedure che costituiscono gli atomi $A_1, \ldots, A_m$.

In accordo a questa visione la valutazione della chiamata $p(t_1, \ldots, t_n.)$ causa la valutazione del corpo della procedura, dopo aver effettuato il passaggio dei parametri in modalita passaggio per nome, rimpiazzando nel corpo della procedura il parametro formale X_i con il corrispondente parametro attuale t_i.

Riassumendo in termini più precisi quanto detto sopra, possiamo dire che la valutazione della chiamata $p(t_1, \ldots, t_n.)$, con la definizione di p vista poc'anzi, causa la valutazione delle m chiamate di procedura $(A_1, \ldots, A_m)\vartheta$ presenti nel corpo di p, opportunamente istanziate dalla sostituzione $\vartheta = \{X_1/t_1, \ldots, X_m/t_m\}$ che effettua il passaggio dei parametri. Nel caso in cui il corpo della clausola sia vuoto (ossia $m = 0$), la chiamata termina immadiatamente, altrimenti la computazione prosegue con la valutazione delle nuove chiamate.

Valutazione di un goal non atomico

Quanto visto in precedenza deve essere generalizzato perché sia chiaro il modello computazionale del paradigma logico.

Nel caso si debba valutare un goal non atomico il meccanismo computazionale è analogo a quello visto sopra, salvo che adesso dobbiamo selezionare una delle chiamate possibili usando un'opportuna regola di selezione. Mentre nel caso della programmazione logica pura non è specificata alcuna regola, PROLOG adotta la regola che seleziona sempre l'atomo più a sinistra, è comunque possibile dimostrare che le risposte calcolate sono sempre lle stesse, indipendentemente da quale sia la regola adottata.

B.4.4 Controllo: il non determinismo

Nella valutazione di un goal abbiamo due gradi di libertà: la selezione dell'atomo da valutare e la scelta della clausola da applicare.

Per il primo abbiamo detto che possimo fissare una regola di selezione, senza che questo influenzi i risultati finali delle computazioni che terminano con successo.

Per la scelta della clausola invece la cosa è più delicata. Dato che un predicato può essere definito da più clausole e dobbiamo usarne una sola alla volta, potremmo pensare di fissare una qualche regola di scelta, analogamente a quanto fatto con la selezione degli atomi. Ma ci si rende subito conto con qualche esempio che, in generale, non esiste un modo di eseguire le clausole che risolva il problema.

Abbiamo introdotto nel modello di computazione una forma di non-determinismo: nel caso in cui vi siano più clausole per lo stesso predicato dobbiamo sceglierne una in modo non determinismistico, senza fissare alcune regola.

Il modello teorico della programmazione logica mantiene questo non determinismo in quanto vengono considerate tutte le possibili scelte delle clausole e quindi tutti i possibili risultati delle varie computazioni prodotte in conseguenza a tali scelte.

Il becktracking in PROLOG

Quando da un modello teorico si passa ad un linguaggio implementato, quale il PROLOG, il non-determinismo ad un qualche livello deve essere trasformato in determinismo, dato le macchine fisiche che usiamo sono deterministiche.

In PROLOG, per motivi di semplicità e di efficienza dell'implementazione, viene usata la strategia in base all'ordine testuale in cui si presentano le clausole all'interno del programma (dall'alto verso il basso). Quindi anche se questa strategia è incompleta (perché si può rischiare di non far terminare mai un'interrogazione), è tuttavia arginabile dal programmatore che, conoscendo questa caratteristica del linguaggio, può ordinare le clausole del programma nel modo più conveniente (tipicamente mettendo prima le regole relative ai casi terminali e poi quelle induttive). Questa tecnica, almeno in linea di principio, elimina parte della dichiaratività del linguaggio.

Oltre allecomputazioni infine, vi è un secondo aspetto, più importante, da considerare adottando il modello deterministico del PROLOG e riguarda la gestione dei fallimenti.

In generale quando si arriva ad un fallimento, la macchina astratta PROLOG viene scelta fa "backtracking" fino al precedente punto di scelta nel quale vi siano altre possibilità.

È facile rendesi conto come questo procedere per tentativi, che sostanzialmente corrisponde ad una ricerca in profondità in un albero che rappresenta tutte le possibili computazioni, può essere molto pesante dal punto di vista computazionale.

B.5 Esempi di computazione in PROLOG

In questo paragrafo faremo riferimento al linguaggio PROLOG, del quale seguiremo anche la sintassi. La notazione $[h|t]$ è usata per indicare la lista che ha come testa h e come coda t. Per testa si intende il primo elemento

della lista, mentre la cosa è la lista costituita dai restanti elementi, una volta tolto il primo. La lista vuota è denotata da [].

Come primo esempio consideriamo il seguente programma MEMBER che verifica se un elemento appartiene ad una lista:

```
member(X, [X | Xs]).
member(X, [Y | Xs]) :- member(X, Xs).
```

La lettura dichiarativa del precedente programma è immediata: la clausola (1) costituisce il caso terminale, in cui l'elemento che stiamo cercando (il primo argomento del predicato member) è la testa della lista che abbiamo (il secondo argomento di member). La clausola (2) fornisce invece il caso induttivo e ci dice che X è un elemento della lista se è un elemento della lista Xs.

Notiamo che questo programma può essere usato in vari modi diversi.

Il modo più convenzionale è quello di usarlo come test: verificare cioè la presenza di un determinato termine all'interno della lista. In questo caso si ha una semplice risposta "booleana" che esprime l'esistenza di una computazione di successo per il nostro goal.

```
?- member(pippo, [pluto, pippo, ciccio]).
Yes
```

Tuttavia possiamo anche usare il programma per calcolare. Ad esempio possiamo chiedere la valutazione di

```
?- member(X, [pluto, pippo, ciccio]).
X = pluto
```

La macchina astratta restituisce {X/pluto} come una risposta calcolata. Possiamo anche calcolare le risposte successive usando il comando ";". Quando non vi sono più risposte il sistema risponde "no".

??(B)

$append \subseteq [\mathcal{P}] \times [\mathcal{P}] \times [\mathcal{P}]$ t.c.:	
$append([], [p], [p])$	append([A], [B], [C])
	append([p, [A]], [B], [p, [C]])

Elenco delle figure

Elenco delle tabelle

Bibliografia

[AA97] Asperti Andrea and Ciabattoni Agata. *Logica a informatica*. McGraw-Hill, 1997.

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive System*. Cambrice university press, 2007.

[Bar85] H. P. Barendregt. *The Lambda Calculus*. studies in logic and the fondations of mathematics. elsevier, 1985.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Inform. and Control*, pages 109–137, 1984.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[FR80] Nissim Francez and Michael Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *FOCS*, pages 373–379, 1980.

[GM90] Roberto Gorrieri and Ugo Montanari. Towards hierarchical description of systems: A proof system for strong prefixing. *Int. J. Found. Comput. Sci.*, 1(3):277–294, 1990.

[GM06] Maurizio Gabbrielli and Simone Martini. *Linguaggi di programmazione*. McGraw-Hill, 2006.

[GMM90] Roberto Gorrieri, Sergio Marchetti, and Ugo Montanari. A2ccks: Atomic actions for ccs. *Theor. Comput. Sci.*, 72(2&3):203–223, 1990.

[Gor12] Roberto Gorrieri. A full-abstract semantics for atomicty. 2012.

[GV10] Roberto Gorrieri and Cristian Versari. A process calculus for expressing finite place/transition petri nets. In *EXPRESS'10*, pages 76–90, 2010.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[LR81] Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] R. Milner. *Comunication and Concurrency*. Prentice-Hall, 1989.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[WC84] Clockisin W.F and Mellish C.S. *programmare in prolog*. INFORMATICA EDP, 1984.

[WN95] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4 of *Lecture Notes in Computer Science*. Clarendon Press, 1995.