Daniele Gorla

Il π-calcolo: nozioni di teoria e applicazioni[1]

Università di Roma "La Sapienza"

Laurea Specialistica in Informatica

Dispense per il Corso di Teoria della Concorrenza

30 aprile 2008

[1]Queste dispense sono una rielaborazione personale di materiale classico sul π-calcolo; il loro scopo è descrivere formalmente il calcolo e le sue basi matematiche, nonchè darne esempi d'uso come modello di varie applicazioni concorrenti e distribuite. Ringrazio gli studenti Benedetta Bergamini, Riccardo Mariani e Nicola Paolino del corso di Teoria della Concorrenza, a.a. 2004-05, per una prima stesura in latex di queste dispense.

Indice

Capitolo 1

Introduzione al π-calcolo

Il π-calcolo [12] nasce nel 1989 all'Università di Edimburgo; esso evolve il CCS [10] aumentandone il potere espressivo e la facilità nel modellare applicazioni concorrenti. Il notevole successo ottenuto dal π-calcolo è testimoniato dalle diverse versioni presentate in letteratura [1, 5, 17, 7, 8, 4, 15] e dalle molteplici implementazioni quali Pict [19, 14, 20], Biztalk [9] o Highwire.

1.1 Dal CCS al π-calcolo

In questo primo capitolo si analizza il motivo per cui si è sentita l'esigenza di andare oltre il CCS introducendo i nuovi concetti che hanno dato vita al π-calcolo. A tal scopo, ricordiamo che il CCS è caratterizzato da due aspetti fondamentali: la *concorrenza* e la *sincronizzazione* tra processi. Pertanto, il CCS formalizza bene computazioni parallele (che si comportano quindi in maniera non deterministica) che possono andare in conflitto per l'uso delle risorse assegnate ai processi stessi.

In queste caratteristiche vi è il punto di forza e nello stesso tempo il punto di debolezza del CCS: infatti, in concorrenza spesso si vorrebbe avere la possibilità di effettuare *comunicazioni*, cioè scambi di messaggi, tra processi. Un nuovo concetto importante che si vorrebbe poter gestire è pertanto lo *scambio di informazioni*.

Che cos'è un messaggio? Si può definire l'oggetto della comunicazione come un elemento di un certo tipo di dato (per esempio un intero, una stringa o un'immagine). Inoltre, come è possibile utilizzare un messaggio? In generale, possiamo identificare due possibili usi: *estensionale* ed *intensionale*. Nel primo caso l'aspetto principale del messaggio è il suo *valore*, da usare per esempio in operazioni logico aritmetiche; questo approccio ha dato vita ad un'evoluzione del CCS chiamata *CCS-value passing* [10]. Nel secondo caso

l'aspetto su cui ci si sofferma sono le funzionalità che il messaggio conferisce al ricevente (ad esempio il messaggio può essere una chiave di cifratura, il cui valore numerico non è fine a sè ma trae la sua ragione d'essere dalle informazioni che è in grado di decifrare). Questo aspetto intensionale è ciò che viene modellato nel π-calcolo.

Quindi, l'idea fondamentale del π-calcolo è che i processi si scambiano tra loro messaggi che portano delle funzionalità. Se nel CCS la computazione è essenzialmente una serie di sincronizzazioni tra processi, nel π-calcolo una computazione è una sequenza di *comunicazioni su canali*. Di conseguenza, una funzionalità è la possibilità di fare una comunicazione che prima non era possibile, magari perchè non si conosceva il canale. Il risultato è che i canali sono allo stesso tempo sia il mezzo di comunicazione che l'oggetto della comunicazione stessa. Ciò determina la differenza principale tra il π-calcolo e il CCS value-passing: in quest'ultimo l'oggetto della comunicazione è un dato fine a se stesso. Tipicamente i canali, che costituiscono il nucleo fondamentale del π-calcolo, vengono chiamati *nomi*.

Esempio 1 Per chiarire intuitivamente le differenze tra CCS, CCS-value passing e π-calcolo, modelliamo uno scenario in cui un cliente vuole comprare una pizza. In CCS, tale situazione è modellabile mettendo in parallelo il cliente, modellato dal processo A, con il pizzaiolo, modellato dal processo B.

$$A \triangleq \overline{askPizza}.\overline{pay}.pizza. < eat\ pizza > \qquad B \triangleq askPizza.pay.\overline{pizza}$$

Intuitivamente, il cliente richiede una pizza, la paga, la riceve e la mangia; viceversa, il pizzaiolo riceve la richiesta di una pizza e i soldi, dopodichè consegna la pizza al cliente. In CCS-value passing, questa semplice interazione può essere arricchita di vari dettagli come, ad esempio, il tipo di pizza richiesto ed il suo prezzo.

$$\begin{aligned} A &\triangleq \overline{askPizza}\langle margherita\rangle.\overline{pay}\langle 4\ Euro\rangle.pizza. < eat\ pizza > \\ B &\triangleq askPizza(x).pay(y).\textbf{if } y = price(x) \textbf{ then } \overline{pizza} \textbf{ else} \\ &\quad \textbf{if } y < price(x) \textbf{ then } \overline{askMoney} \textbf{ else } \overline{pizza}.\overline{output}\langle y - price(x)\rangle \end{aligned}$$

Infine nel π-calcolo possiamo modellare bene la consegna a domicilio della pizza.

$$\begin{aligned} A &\triangleq \overline{askPizza}\langle myHome\rangle.myHome(x).\overline{pay}. < eat\ x > \\ B &\triangleq askPizza(y).\overline{y}\langle pizza\rangle.pay \end{aligned}$$

Calcolo per la gestione dei nomi Cosa è possibile fare con un nome? Lo si può spedire, lo si può usare per comunicare o se ne può testare il valore; inoltre, in pratica, è spesso utile creare dei nomi nuovi, o *ristretti*, diversi da tutti i nomi esistenti. Per esempio, ciò può essere utile per proteggere il contenuto dei messaggi inviati e modellare quindi dei canali sicuri. Questi nomi nuovi possono essere visti come delle variabili locali di un linguaggio imperativo, ma che, al pari degli altri nomi, possono essere passati nel corso di una comunicazione. Pertanto, la gestione dello scopo di un nome ristretto è una questione delicata.

Per esempio immaginiamo una comunicazione tre due utenti, Alice e Bob, con Alice che crea un nome ristretto per comunicare a Bob informazioni riservate.

$$A \triangleq (\nu c)\overline{a}\langle c\rangle.\overline{c}\langle\cdots\rangle \qquad\qquad B \triangleq a(x).x(\cdots)$$

Dopo la creazione e la trasmissione del nuovo canale c, Alice è pronta a comunicare su c (si evolve, cioè, in $A' \triangleq \overline{c}\langle\cdots\rangle$) e Bob è pronto a ricevere da c (si evolve, cioè, in $B' \triangleq c(\cdots)$). Infatti, a seguito della comunicazione, ogni occorrenza di x in B viene sostituita da c. Pertanto, l'interazione tra Alice e Bob porta a

$$A \mid B \longmapsto (\nu c)(\overline{c}\langle\cdots\rangle \mid c(\cdots))$$

Da notare che lo scopo di c prima della comunicazione include solo Alice ma dopo include anche Bob. Quindi si può dire che lo scopo di un nome non è fissato sintatticamente come nei linguaggi imperativi ma può variare a seguito di comunicazioni.

1.2 Presentazione Formale del π-calcolo

1.2.1 Sintassi del π-calcolo

Assumiamo un insieme numerabile di *nomi* $\mathcal{N}$. Come convenzione, le prime lettere dell'alfabeto $a, b, c, ...$ vengono usate per indicare nomi noti, mentre $x, y, z, ...$ indicano variabili di input; queste sono dei segnaposti che verranno poi istanziati da altri nomi a seguito di comunicazioni. Un processo del π-calcolo è definito come:

$$P \;::=\; \mathbf{0} \;\mid\; a(x).P \;\mid\; \overline{a}\langle b\rangle.P \;\mid\; P_1|P_2 \;\mid\; (\nu a)P \;\mid\; [a=b]P \mid\; !P$$

dove

- $\mathbf{0}$ è il processo inattivo, che non compie nessuna azione;

- $a(x)$ indica l'input mentre $\overline{a}\langle b\rangle$ indica l'output (essi sono l'equivalente delle azioni e coazioni del CCS, con in più la possibilità di emettere/ricevere un valore su/da un canale);
- '|' è l'operatore di composizione parallela, con semantica ad interleaving;
- $(\nu a)P$ rende a locale a P, cioè ne restringe lo scopo al processo P;
- $[a = b]P$ rappresenta il name matching ed è l'equivalente del costrutto **if** $a = b$ **then** P;
- $!P$ modella la replicazione ed indica un numero arbitrario di copie di P in parallelo.

Va notato che, differentemente dal CCS, la ricorsione viene modellata tramite replicazione e che non c'è l'operatore di scelta non-deterministica. Mostreremo che la prima differenza è irrilevante, visto che definizioni ricorsive e parametriche di processi possono essere implementate nel calcolo che abbiamo appena presentato (vedi Esempio 5 più avanti). Per una discussione approfondita sul potere espressivo della scelta nell'ambito del π-calcolo, si rimanda a [13]; qui diciamo solo che, per gli argomenti che andremo a trattare nel seguito, la presenza o l'assenza di tale operatore è irrilevante.

Nei processi $a(x).P$ e $(\nu a)P$, i nomi x e a sono detti *legati* (o *bound*) e P è lo scopo di tali nomi. Un nome che non è legato è detto *libero* (o *free*). Definiamo quindi $fn(P)$ e $bn(P)$ come gli insiemi dei nomi liberi e legati di P; denotiamo con $n(P)$ tutti i nomi di P (cioè, $n(P) = fn(P) \cup bn(P)$). L'*alfa-conversione*, denotata con $=_\alpha$, permette di ridenominare un nome legato con un nome nuovo (o *fresh*); essa permette, cioè, di rimpiazzare ogni occorrenza legata di un nome con un'occorrenza di un altro nome che non occorre già nel processo in questione. Ad esempio, $(\nu d)\overline{a}\langle d\rangle.a(y).y(z).\mathbf{0}$ è ottenibile per alfa-conversione da $(\nu c)\overline{a}\langle c\rangle.a(x).x(z).\mathbf{0}$.

Per migliorare la leggibilità, le occorrenze di $\mathbf{0}$ nella maggior parte dei casi vengono sottintese; ad esempio, il processo $\overline{a}\langle b\rangle.\mathbf{0}$ viene abbreviato come $\overline{a}\langle b\rangle$. Nel resto di queste note useremo sempre questa convenzione.

1.2.2 Semantica a Riduzioni per il π-calcolo

Per dare una semantica operazionale al π-calcolo si possono seguire due approcci:

- tramite un sistema di transizioni etichettate (LTS, dall'inglese Labelled Transition System)

- tramite una semantica a riduzioni

Nel primo approccio l'idea è di descrivere le interazioni che un processo offre all'esterno descrivendo tutto ciò che può fare un termine. L'LTS dà una semantica "esaustiva" nel senso che analizza tutti i comportamenti di un processo (sia quelli attuali che quelli potenziali); per far questo, ha almeno una regola per ogni costrutto sintattico. Si osservino, ad esempio, le regole del parallelo in CCS:

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

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

Pertanto, il termine $a.P|\overline{a}.Q$ ha tre possibili evoluzioni:

$$a.P \mid \overline{a}.Q \quad \begin{array}{ll} \xrightarrow{a} & P \mid \overline{a}.Q \\ \xrightarrow{\bar{a}} & a.P \mid Q \\ \xrightarrow{\tau} & P \mid Q \end{array}$$

La semantica a riduzioni invece descrive solamente le computazioni attuali di un termine, cioè quelle che un proceso può generare da solo (in quanto al suo interno ci sono già le potenzialità per evolvere); quindi modella solo i τ dell'LTS. Per esempio, il processo $a.P$ non ha riduzioni poichè è bloccato in attesa di sincronizzarsi con qualche altro processo su a. Concentrandosi solamente sui comportamenti attuali, la semantica a riduzioni è meno descrittiva ma minimizza il numero di regole: l'idea è quella di cercare il numero minimo di regole operazionali e di identificare un'opportuna equivalenza tra processi che raggruppi processi con le stesse riduzioni.

Partiamo con il definire una *equivalenza strutturale* tra processi, cioè un'equivalenza tra processi che "hanno la stessa struttura", nel senso che descrivono lo stesso sistema concorrente[1]. Gli assiomi e le regole per tale equivalenza sono riportati in Tabella 1.1. L'assioma (S-Conv) uguaglia processi alfa-convertibili; gli assiomi (S-Id), (S-Com) e (S-Ass) dicono che il parallelo è un operatore monoidale, con $\mathbf{0}$ come identità; (S-Eq) dice che il test di uguaglianza tra due nome uguali è sempre passato positivamente; (S-Rep)

[1]La sintassi del π-calcolo è infatti solo un modo per scrivere sistemi concorrenti. Pertanto, lo stesso sistema può essere descritto in modi diversi (ma equivalenti); si considerino, ad esempio, i due processi $P|Q$ e $Q|P$. Lo scopo dell'equivalenza strutturale è di rendere irrilevanti tali differenze.

(S-Conv) $P \equiv P' \quad$ se $P =_\alpha P'$	(S-Id) $P \mid \mathbf{0} \equiv P$
(S-Com) $P \mid Q \equiv Q \mid P$	(S-Ass) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
(S-Eq) $[a = a]P \equiv P$	(S-Rep) $!P \equiv P \mid !P$
(S-Abs) $(\nu a)\mathbf{0} \equiv \mathbf{0}$	(S-Ext) $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad$ se $a \notin fn(P)$
(S-RCom) $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$	(S-Refl) $P \equiv P$
(S-Simm) $\dfrac{P \equiv Q}{Q \equiv P}$	(S-Trans) $\dfrac{P \equiv Q \quad Q \equiv R}{P \equiv R}$
(S-Par) $\dfrac{P \equiv P'}{P \mid Q \equiv P' \mid Q}$	(S-Res) $\dfrac{P \equiv P'}{(\nu a)P \equiv (\nu a)P'}$

Tabella 1.1: Equivalenza Strutturale

permette di attivare una copia parallela del processo replicato; (S-Abs) dice che, se un nome ristretto ha come scopo il processo vuoto, allora tale nome è inutile; (S-RCom) dice che l'ordine con cui vengono ristretti i nomi non è rilevante[2]. La regole (S-Refl), (S-Simm) e (S-Trans) dicono che $\equiv$ è una equivalenza, mentre (S-Par) e (S-Res) rendono $\equiv$ una congruenza (cioè un'equivalenza chiusa per contesti). Infine, l'assioma (S-Ext) serve per estendere lo scopo di un nome ristretto, purchè tale nome non compaia nel processo che si vuole includere nello scopo; ciò è necessario per preservare lo stato di nome nuovo di ogni nome ristretto.

Esempio 2 Per chiarire l'importanza della side condition della regola (S-Ext), si consideri il processo

$$Q \triangleq (\nu b)(\bar{b}\langle c\rangle | b(x_1).Q_1 | ... | b(x_n).Q_n)$$

In Q, c è una risorsa che può essere usata solo da $Q_1, ..., Q_n$. Consideriamo ora il processo $P \triangleq b(x).P'$; mettendo P in parallelo a Q e usando (S-Ext)

[2]Per via di (S-RCom), il processo $(\nu a)(\nu b)P$ verrà spesso scritto come $(\nu a, b)P$; più in generale, $(\nu a_1)\cdots(\nu a_n)P$ verrà scritto come $(\nu \widetilde{a})P$, dove $\widetilde{a}$ denota l'insieme $\{a_1, \cdots, a_n\}$.

(R-Com)	$a(x).P \mid \overline{a}\langle b\rangle.Q \longmapsto P[^b/_x] \mid Q$
(R-Par)	$\dfrac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q}$
(R-Res)	$\dfrac{P \longmapsto P'}{(\nu a)P \longmapsto (\nu a)P'}$
(R-Struct)	$\dfrac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}$

Tabella 1.2: Riduzioni per il π-calcolo

senza la side-condition $b \notin fn(P)$, si potrebbe avere:

$$P|Q \equiv (\nu b)(\overline{b}\langle c\rangle|b(x_1).Q_1|...|b(x_n).Q_n|b(x).P')$$

e quindi c potrebbe essere intercettato da P'.

Passiamo ora alle riduzioni; esse sono date in Tabella 1.2. Come già detto, l'essenza del π-calcolo sta nella comunicazione tra processi. Pertanto, la regola base della semantica è (R-Com); questa regola indica che se si spedisce b su a e qualcuno è pronto a ricevere da a, allora avviene una comunicazione. A seguito di ciò, i prefissi di input e di output vengono consumati; inoltre, il processo che ha ricevuto b sostituisce con tale nome ogni occorrenza della variabile di input x. La composizione parallela di processi è modellata dalla regola (R-Par). Questa regola modella una semantica ad interleaving nel senso che, se una comunicazione avviene in un sottoprocesso, allora l'intero processo evolve di conseguenza. La situazione è simile se la comunicazione avviene all'interno dello scopo di una restrizione, come descritto dalla regola (R-Res). Infine, (R-Struct) stabilisce che processi strutturalmente equivalenti hanno le stesse riduzioni.

Per concludere, mostriamo un semplice esempio che mostra come usare le alcune delle regole operazionali appena discusse. Assumiamo che $b \notin fn(P)$; allora, usando (R-Struct), (R-Res) e (R-Com) otteniamo

$$\dfrac{a(x).P|(\nu b)\overline{a}\langle b\rangle.Q \;\equiv\; \dfrac{a(x).P|\overline{a}\langle b\rangle.Q \longmapsto P[^b/_x]|Q}{(\nu b)(a(x).P|\overline{a}\langle b\rangle.Q) \longmapsto (\nu b)(P[^b/_x]|Q)}}{a(x).P|(\nu b)\overline{a}\langle b\rangle.Q \longmapsto (\nu b)(P[^b/_x]|Q)}$$

1.3 Esempi

Passiamo ora a presentare alcuni esempi, al fine di illustrare le potenzialità del π-calcolo nel modellare sistemi concorrenti.

Esempio 3 Cominciamo con il mostrare come l'utilizzo di canali ristretti possa evitare interferenze. Immaginiamo di voler fare campagna elettorale per un determinato candidato e di voler evitare ingerenze esterne. Consideriamo i seguenti processi:

$$\begin{aligned} Speaker &\triangleq \overline{air}\langle vota\ pippo\rangle \\ Microfono &\triangleq air(x).\overline{wire}\langle x\rangle \\ Altoparlante &\triangleq wire(y).\overline{highvolume}\langle y\rangle \\ Comozio &\triangleq Speaker \mid Microfono \mid Altoparlante \end{aligned}$$

Tale sistema evolve come segue:

$$\begin{aligned} Comozio &\longmapsto \overline{wire}\langle vota\ pippo\rangle \mid Altoparlante \\ &\longmapsto \overline{highvolume}\langle vota\ pippo\rangle \end{aligned}$$

Si consideri ora un possibile intervento di un rivale

$$Rivale \triangleq wire(z).\overline{wire}\langle vota\ pluto\rangle$$

Si osservi cosa potrebbe succedere quando i processi *Comizio* e *Rivale* procedono in parallelo:

$$\begin{aligned} Comizio \mid Rivale &\longmapsto \overline{wire}\langle vota\ pippo\rangle \mid Altoparlante \mid Rivale \\ &\longmapsto \overline{wire}\langle vota\ pluto\rangle \mid Altoparlante \\ &\longmapsto \overline{highvolume}\langle vota\ pluto\rangle \end{aligned}$$

Il problema è che il rivale si è potuto intromettere nel comizio poichè *wire* è un canale pubblico, accessibile a tutti. Definiamo ora un nuovo processo in cui *air* e *wire* sono nomi ristretti:

$$ComizioSicuro \triangleq (\nu\, air, wire)(Speaker \mid Microfono \mid Altoparlante)$$

Tale processo è sicuro, nel senso che il rivale non è più in grado di interferire nel comizio.

Esempio 4 Immaginiamo ora due processi, Alice e Bob, che vogliono stabilire un canale segreto utilizzando un Server con cui hanno ognuno un canale riservato. Definiamo:

- c_{AS} il canale di comunicazione segreto tra A e S
- c_{BS} il canale di comunicazione segreto tra B e S
- c_{AB} il canale di comunicazione segreto da stabilire tra A e B.

Si possono quindi programmare Alice, Bob e il Server nel modo seguente:

$$\begin{array}{rcl} A & \triangleq & (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle mess\rangle \\ S & \triangleq & !c_{AS}(x).\overline{c_{BS}}\langle x\rangle \\ B & \triangleq & c_{BS}(z).z(w).< usa\ w > \end{array}$$

Il sistema risultante evolve come segue:

$$(\nu c_{AS}, c_{BS})(A|S|B) \longmapsto\longmapsto\longmapsto (\nu c_{AS}, c_{BS}, c_{AB})(S \mid\ < usa\ mess >)$$

E' evidente che la comunicazione è senza interferenze poichè i canali sono ristretti.

Esempio 5 È spesso utile, nello scrivere processi, descrivere comportamenti ricorsivi similmente all'invocazione di procedure (ricorsive) con passaggio di parametri. Sarebbe, cioè, utile estendere la sintassi del π-calcolo con *identificatori di processi*, ad esempio $A, B, C, ...$, ognuno associato ad un'unica definizione di processo ed un'arità (il numero di parametri passati nell'invocare il processo associato). Ad esempio,

$$SendAlong(x_1, x_2, x_3) \ \triangleq\ \overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle.SendAlong\langle x_1, x_2, x_3\rangle$$

identifica un processo che spedisce ripetutamente i nomi x_2 e x_3 sul canale x_1.

L'uso di tale sintassi estesa è solo notazionalmente più utile poichè, come ora mostreremo, l'uso di identificatori di processo per definire ed invocare ricorsivamente processi con parametri è realizzabile nel π-calcolo presentato finora tramite replicazione, restrizione e comunicazioni.

Associamo ad ogni identificatore di processo A di arità k un canale riservato c_A. La definizione di processo $A(x_1, \cdots, x_k) \triangleq P$ diventa il processo

$$!\, c_A(x).x(x_1).\cdots.x(x_k).P$$

per x fresh, che verrà messo in parallelo con i restanti termini. Ogni invocazione $A\langle b_1, \cdots, b_k\rangle$ viene invece sostituita col sotto-processo

$$(\nu a)\overline{c_A}\langle a\rangle.\overline{b}\langle b_1\rangle.\cdots.\overline{b}\langle b_k\rangle$$

Per esempio, il processo $SendAlong\langle b, c, d\rangle$ viene codificato come segue:

$$\begin{array}{l} !\, c_{SendAlong}(x).x(x_1).x(x_2).x(x_3).\overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle. \\ \quad (\nu a)\overline{c_{SendAlong}}\langle a\rangle.\overline{a}\langle x_1\rangle.\overline{a}\langle x_2\rangle.\overline{a}\langle x_3\rangle \\ \mid (\nu a)\overline{c_{SendAlong}}\langle a\rangle.\overline{a}\langle b\rangle.\overline{a}\langle c\rangle.\overline{a}\langle d\rangle \end{array}$$

Tale processo, dopo 4 riduzioni, diventa

$$\begin{array}{l} !\, c_{SendAlong}(x).x(x_1).x(x_2).x(x_3).\overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle. \\ \quad (\nu a)\overline{c_{SendAlong}}\langle a\rangle.\overline{a}\langle x_1\rangle.\overline{a}\langle x_2\rangle.\overline{a}\langle x_3\rangle \\ \mid \overline{b}\langle c\rangle.\overline{b}\langle d\rangle.(\nu a)\overline{c_{SendAlong}}\langle a\rangle.\overline{a}\langle b\rangle.\overline{a}\langle c\rangle.\overline{a}\langle d\rangle \end{array}$$

Tale processo si comporta esattamente come (cioè, è fortemente bisimile a) $SendAlong\langle b, c, d\rangle$.

Esempio 6 Il π-calcolo serve, come si è detto, a scrivere processi concorrenti; ciononostante, può essere usato per programmare altri tipi di applicazione. Per esempio, mostreremo ora come sia possibile modellare in esso i numeri naturali e le ovvie operazioni su di essi.

Tradizionalmente abbiamo che

$$n = succ^n(0) = 0 + \underbrace{1 + 1 + 1 ... + 1}_{n \ volte}$$

Nel π-calcolo ciò può essere modellato come segue

$$\underline{n}_a \triangleq \overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle.\overline{a}\langle z\rangle$$

dove z e u sono nomi riservati che rappresentano rispettivamente 0 e 1. Questo processo spedisce n volte 1 e poi lo 0 finale. Con tale rappresentazione, la funzione successore è definita come segue:

$$Succ(a, b) \ \triangleq \ a(x).([x = z]\overline{b}\langle u\rangle.\overline{b}\langle z\rangle \mid \ [x = u]\overline{b}\langle u\rangle.Succ(a, b))$$

Dimostriamo ora che l'implementazione nel π-calcolo della funzione successore è corretta. A tale scopo dimostriamo che

$$(\nu a)(Succ(a, b) \mid \underline{n}_a) \ \approx \ \underline{n + 1}_b$$

dove $\approx$ è la bisimulazione debole nel π-calcolo (vedi Sezione 1.5). La dimostrazione si svolge per induzione su n.

Passo Base $(n = 0)$**:**

$$\begin{aligned}
&(\nu a)(Succ(a,b) \mid \overline{a}\langle z\rangle)\\
&\quad \approx\ (\nu a)([z=z]\overline{b}\langle u\rangle.\overline{b}\langle z\rangle \mid\ [z=u]\overline{b}\langle u\rangle.Succ(a,b))\\
&\quad \equiv\ \overline{b}\langle u\rangle.\overline{b}\langle z\rangle\ \triangleq\ \underline{1}_b
\end{aligned}$$

Passo Induttivo:

$$\begin{aligned}
&(\nu a)(Succ(a,b) \mid \overbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}^{n+1}.\overline{a}\langle z\rangle)\\
&\quad \approx\ (\nu a)([u=u]\overline{b}\langle u\rangle.Succ(a,b) \mid \overbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}^{n}.\overline{a}\langle z\rangle)\\
&\quad \equiv\ (\nu a)(\overline{b}\langle u\rangle.Succ(a,b)|\underbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}_{n}.\overline{a}\langle z\rangle)\\
&\quad \approx\ \overline{b}\langle u\rangle.(\nu a)(Succ(a,b) \mid \underline{n}_a)\\
&\quad \approx\ \overline{b}\langle u\rangle.\underline{n+1}_b\ \triangleq\ \overline{b}\langle u\rangle.\underbrace{\overline{b}\langle u\rangle.\cdots.\overline{b}\langle u\rangle}_{n+1}.\overline{b}\langle z\rangle\ \triangleq\ \underline{n+2}_b
\end{aligned}$$

Esercizio: Si definisca il processo $Add(a,b,c)$ per sommare i naturali codificati sui canali a e b che emette il risultato sul canale c. Si mostri poi, ad esempio, che $(\nu a,b)(Add(a,b,c) \mid \underline{1}_a \mid \underline{2}_b) \approx \underline{3}_c$.

Esempio 7 Come ultimo esempio daremo due diverse implementazioni dei booleani per il π-calcolo. La prima implementazione usa due nomi riservati: tt e ff che corrispondono ai due valori di verità. Quindi $True_a$ e $False_a$ sono due processi che spediscono tt e ff su a:

$$True_a \triangleq \overline{a}\langle tt\rangle \qquad\qquad False_a \triangleq \overline{a}\langle ff\rangle$$

Di conseguenza, un processo per testare un valore booleano (stile if-then-else) è

$$Test_a(P;Q)\ \triangleq\ a(x).([x=tt]P \mid\ [x=ff]Q)$$

Ora è facile dimostrare che

$$True_a|Test_a(P;Q) \longmapsto P \mid\ [tt=ff]Q\ \approx\ P$$

La seconda implementazione non usa nè nomi riservati nè matching. L'idea è quella di passare a $True$ e $False$ due nomi; $True$ risponde sul primo mentre $False$ sul secondo:

$$True_a \triangleq a(x).x(y).x(z).\overline{y}\langle y\rangle \qquad\qquad False_a \triangleq a(x).x(y).x(z).\overline{z}\langle z\rangle$$

Quindi, per testare un booleano, bisogna passare due nomi, c e d, e vedere su quale dei due si avrà risposta:

$$Test_a(P;Q) \triangleq (\nu b,c,d)\overline{a}\langle b\rangle.\overline{b}\langle c\rangle.\overline{b}\langle d\rangle.(c(x).P \mid d(x).Q)$$

dove b,c,d e x non occorrono in P e Q, e sono tra loro distinti. Per esempio

$$\begin{aligned} &Test_a(P,Q) \mid True_a \\ &\quad \longmapsto (\nu b)(b(y).b(z).\overline{y}\langle y\rangle \mid (\nu c,d)\overline{b}\langle c\rangle.\overline{b}\langle d\rangle.(c(x).P \mid d(x).Q)) \\ &\quad \longmapsto\longmapsto (\nu b,c,d)(\overline{c}\langle c\rangle \mid c(x).P \mid d(x).Q) \\ &\quad \longmapsto (\nu b,c,d)(P \mid d(x).Q) \approx P \end{aligned}$$

Si noti che i due nomi devono essere trasmessi su un nome ristretto b per evitare interferenze. L'implementazione sarebbe più efficiente potendo passare i due nomi simultaneamente a $True$ e a $False$ invece che uno alla volta (vedi Esempio 19 in Sezione 2.1).

1.4 Un LTS per il π-calcolo

Il grande vantaggio della semantica a riduzioni è la sua semplicità. Come spesso accade, il suo punto di forza è anche il suo punto debole. Infatti, il concentrarsi solo sulle computazioni "attuali", cioè sulle comunicazioni realmente avvenute, ci fa perdere dell'informazione. In particolare, le riduzioni considerano il comportamento di un processo "in isolamento", trascurando le possibili interazioni a cui esso può partecipare, una volta messo in parallelo con altri processi. Questo aspetto risulta fondamentale per sviluppare una semantica "composizionale", in cui cioè il comportamento di un processo è interamente descritto a partire dai comportamenti delle sue componenti.

Pertanto, mostreremo ora una semantica per il π-calcolo alternativa (ma equivalente) alle riduzioni tramite un sistema di transizioni etichettate. Su tale LTS studieremo infine la nozione di bisimulazione per il π-calcolo. Visto che la sintassi del π-calcolo deriva dal CCS, ci aspettiamo che anche il suo LTS deriverà da quello del CCS; tuttavia, l'LTS per il π-calcolo deve gestire aspetti cruciali: il passaggio di nomi e la distinzione tra nomi ristretti e liberi. Nel far ciò, ovviamente, dovrà rispettare la semantica definita da $\longmapsto$.

Gli assiomi e le regole che definiscono la relazione di transizione etichettata tra processi, $\xrightarrow{\alpha}$, sono in Tabella 1.3 dove assumiamo sottintesa una regola che assegna stesse transizioni a processi alfa-convertibili. Le regole (LTS-Out) e (LTS-In) sono la generalizzazione delle regole per azioni e coazioni del CCS. Si noti che in (LTS-In) la variabile di input è già instanziata con il nome che sarà passato nella comunicazione. Pertanto, la regola

(LTS-In)	$a(x).P \xrightarrow{ab} P[b/x]$
(LTS-Out)	$\overline{a}\langle b\rangle.P \xrightarrow{\bar{a}b} P$
(LTS-Com)	$\dfrac{P \xrightarrow{ab} P' \qquad Q \xrightarrow{\bar{a}b} Q'}{\begin{array}{c} P \mid Q \xrightarrow{\tau} P' \mid Q' \\ Q \mid P \xrightarrow{\tau} Q' \mid P' \end{array}}$
(LTS-Res)	$\dfrac{P \xrightarrow{\alpha} P'}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'} \quad b \notin n(\alpha)$
(LTS-Open)	$\dfrac{P \xrightarrow{\bar{a}b} P'}{(\nu b)P \xrightarrow{\bar{a}(b)} P'} \quad a \neq b$
(LTS-Close)	$\dfrac{P \xrightarrow{ab} P' \qquad Q \xrightarrow{\bar{a}(b)} Q'}{\begin{array}{c} P \mid Q \xrightarrow{\tau} (\nu b)(P' \mid Q') \\ Q \mid P \xrightarrow{\tau} (\nu b)(Q' \mid P') \end{array}} \quad b \notin fn(P)$
(LTS-Par)	$\dfrac{P \xrightarrow{\alpha} P'}{\begin{array}{c} P \mid Q \xrightarrow{\alpha} P' \mid Q \\ Q \mid P \xrightarrow{\alpha} Q \mid P' \end{array}} \quad bn(\alpha) \cap fn(Q) = \emptyset$
(LTS-Eq)	$\dfrac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'}$
(LTS-Rep)	$\dfrac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$

Tabella 1.3: LTS per il π-calcolo

(LTS-Com) generalizza in maniera ovvia la regola di sincronizzazione del CCS. Le regole (LTS-Rep) e (LTS-Eq) gestiscono le riduzioni di processi replicati o condizionati da un test su nomi verificato.

Veniamo ora alla parte più delicata dell' LTS, cioè la gestione dei nomi ristretti. Dalla semantica a riduzioni sappiamo che $a(x).P \mid (\nu b)\overline{a}\langle b\rangle.Q$ si riduce a $(\nu b)(P[b/x] \mid Q)$, purchè $b \notin fn(P)$. Per capire come modellare questo comportamento nell' LTS dobbiamo capire come gestire un processo che inizia con una restrizione. A tale scopo, prendiamo in considerazione una

generalizzazione della regola (R-Res)

$$\frac{P \xrightarrow{\alpha} P'}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'} \qquad (\star)$$

ed analizziamone il comportamento in vari casi:

$b \notin n(\alpha)$: In tal caso la regola $(\star)$ può essere usata senza problemi, vedi (LTS-Res).

$\alpha = bc$ o $\alpha = \overline{b}\langle c\rangle$: In tal caso, P fa un input/output lungo il canale b, che è ristretto. Chiaramente, è inutile mostrare all'esterno questa azione, in quanto il canale b è conosciuto solo da P e P non potrà sincronizzarsi in nessun modo con un qualche altro processo su tale canale; pertanto, in questo caso la regola $(\star)$ non vale.

$\alpha = ab$ per $a \neq b$: In questo caso, il canale è libero; quindi un eventuale processo in parallelo a P può passargli messaggi su a. Il problema è che, essendo b creato da P, quest'ultimo non può ricevere b dall'esterno. Quindi anche in questo caso la regola $(\star)$ non è applicabile.

$\alpha = \overline{a}\langle b\rangle$ per $a \neq b$: In questo caso, non ci sono conflitti tra la restrizione di b e l'etichetta: P esporta su un canale libero un nome ristretto. Qui il problema è che, a seguito della ricezione di tale nome, il ricevente deve essere incluso nello scopo della restrizione. Assumere che $(\nu b)P$ evolve in $(\nu b)P'$ rende ciò impossibile; quindi, anche in questo caso la regola $(\star)$ non è applicabile.

Quello che vorremmo in definitiva è una regola che mentre facciamo la comunicazione estenda anche lo scopo del nome. La regola (LTS-Open) risolve questo problema rimuovendo la restrizione davanti a P' e introducendo una nuova etichetta $\bar{a}(b)$, la cui finalità è di memorizzare che b è ristretto. Intuitivamente, quando spediamo un nome ristretto spostiamo la restrizione dal processo all'etichetta (come se "aprissimo" temporaneamente lo scopo del nome). Viceversa, quando avviene una comunicazione di un nome ristretto, ripassiamo la restrizione dall'etichetta al processo (cioè, "chiudiamo" lo scopo del nome includendo il ricevente, vedi (LTS-Close)). Chiaramente nel far ciò dobbiamo evitare di restringere nomi che originariamente erano liberi; è questo lo scopo della side condition alla regola (LTS-Close).

L'ultima regola dell'LTS è (LTS-Par), che generalizza (R-Par). Si noti l'importanza della side condition quando α è un bound output, diciamo $\bar{a}(b)$.

In tal caso, dobbiamo garantire che b non compaia libero in Q; infatti, a seguito di una comunicazione, la regola (LTS-Close) includerebbe anche Q nello scopo della restrizione su b, rendendo così ristrette delle occorrenze di b in Q che prima erano libere. Per esempio, consideriamo il processo $(P \mid Q) \mid R$, dove

$$P \triangleq (\nu b)\overline{a}\langle b\rangle.P' \qquad Q \triangleq b(x).Q' \qquad R \triangleq a(x).R'$$

Senza la side condition alla regola (LTS-Par), avremmo che

$$P \mid Q \xrightarrow{\bar{a}(b)} P' \mid Q$$

e quindi

$$(P \mid Q) \mid R \xrightarrow{\tau} (\nu b)(P' \mid Q \mid R'[b/x])$$

Questo farebbe sì che l'occorrenza (libera) di b in Q nel processo $(P \mid Q) \mid R$ diventi legata a seguito della comunicazione. La side condition alla regola (LTS-Par) impone di alfa-convertire b con un nome nuovo (diciamo b') in P prima di spedirlo e di far evolvere $(P \mid Q) \mid R$ in $(\nu b')(P'[b'/b] \mid Q \mid R'[b'/x])$.

Per concludere la presentazione dell'LTS dimostriamo che la semantica da esso indotta coincide con la semantica a riduzioni. A tale scopo, partiamo con un lemma tecnico che mette in corrispondenza le etichette generate da un processo con la sua struttura sintattica. La prova (per induzione sulla lunghezza dell'inferenza della transizione etichettata) è lasciata per esercizio al lettore.

Lemma 8

1. *Se* $P \xrightarrow{ab} P'$, *allora* $P \equiv (\nu\widetilde{a})(a(x).P_1 \mid P_2)$ *per* $a, b \notin \widetilde{a}$ *e* $P' \equiv (\nu\widetilde{a})(P_1[b/x] \mid P_2)$.

2. *Se* $P \xrightarrow{\bar{a}b} P'$, *allora* $P \equiv (\nu\widetilde{a})(\overline{a}\langle b\rangle.P_1 \mid P_2)$ *per* $a, b \notin \widetilde{a}$ *e* $P' \equiv (\nu\widetilde{a})(P_1 \mid P_2)$.

3. *Se* $P \xrightarrow{\bar{a}(b)} P'$, *allora* $P \equiv (\nu\widetilde{a}, b)(\overline{a}\langle b\rangle.P_1 \mid P_2)$ *per* $a \notin \widetilde{a}, b$ *e* $P' \equiv (\nu\widetilde{a})(P_1 \mid P_2)$.

4. *Se* $P_1 \mid P_2 \xrightarrow{\alpha} P$, *allora*

   (a) $P_1 \xrightarrow{\alpha} P_1'$ *e* $P \triangleq P_1' \mid P_2$; *oppure*

   (b) $P_2 \xrightarrow{\alpha} P_2'$ *e* $P \triangleq P_1 \mid P_2'$; *oppure*

   (c) $P_1 \xrightarrow{ab} P_1'$, $P_2 \xrightarrow{\bar{a}b} P_2'$ *e* $P \triangleq P_1' \mid P_2'$, *con* $\alpha = \tau$; *oppure*

   (d) $P_1 \xrightarrow{\bar{a}b} P_1'$, $P_2 \xrightarrow{ab} P_2'$ *e* $P \triangleq P_1' \mid P_2'$, *con* $\alpha = \tau$; *oppure*

(e) $P_1 \xrightarrow{ab} P_1'$, $P_2 \xrightarrow{\bar{a}(b)} P_2'$ *e* $P \triangleq (\nu b)(P_1' \mid P_2')$, *con* $\alpha = \tau$; *oppure*

(f) $P_1 \xrightarrow{\bar{a}(b)} P_1'$, $P_2 \xrightarrow{ab} P_2'$ *e* $P \triangleq (\nu b)(P_1' \mid P_2')$, *con* $\alpha = \tau$.

Proposizione 9 *Se* $P \xrightarrow{\tau} P'$, *allora* $P \longmapsto P'$.

Dim.: per induzione sulla lunghezza dell'inferenza di $P \xrightarrow{\tau} P'$.

Passo Base: le uniche regole in cui il τ appare solo nella conclusione sono (LTS-Com) e (LTS-Close). Consideriamo (LTS-Com). In tal caso, $P \triangleq P_1 \mid P_2$ e $P_1' \mid P_2' \triangleq P'$, poichè $P_1 \xrightarrow{ab} P_1'$ e $P_2 \xrightarrow{\bar{a}b} P_2'$. Per il Lemma 8(1)/(2), $P_1 \equiv (\nu\widetilde{a})(a(x).P_3 \mid P_4)$, $P_1' \equiv (\nu\widetilde{a})(P_3[b/x] \mid P_4)$, $P_2 \equiv (\nu\widetilde{b})(\overline{a}\langle b\rangle.P_5 \mid P_6)$ e $P_2' \equiv (\nu\widetilde{b})(P_5 \mid P_6)$. Sotto queste ipotesi, usando (R-Com), (R-Par), (R-Res) e (R-Struct), possiamo trovare una riduzione da P a P':

$$\dfrac{\dfrac{\dfrac{a(x).P_3 \mid \overline{a}\langle b\rangle.P_5 \longmapsto P_3[b/x] \mid P_5}{a(x).P_3 \mid \overline{a}\langle b\rangle.P_5 \mid P_4 \mid P_6 \longmapsto P_3[b/x] \mid P_5 \mid P_4 \mid P_6}}{(\nu\widetilde{a},\widetilde{b})(a(x).P_3 \mid \overline{a}\langle b\rangle.P_5 \mid P_4 \mid P_6) \longmapsto (\nu\widetilde{a},\widetilde{b})(P_3[b/x] \mid P_5 \mid P_4 \mid P_6)}}{P \longmapsto P'}$$

Il caso per (LTS-Close) procede nello stesso modo, ma usa il terzo caso del Lemma 8 invece del secondo.

Passo Induttivo: distinguiamo l'ultima regola usata.

(LTS-Par): in questo caso, $P \triangleq P_1 \mid P_2$ e $P' \triangleq P_1' \mid P_2$, dove $P_1 \xrightarrow{\tau} P_1'$. Per induzione, $P_1 \longmapsto P_1'$ e, per (R-Par), $P \longmapsto P'$.

(LTS-Res): simile a (LTS-Par).

(LTS-Rep): in questo caso, $P \triangleq !P_1$ e $P_1 \mid !P_1 \xrightarrow{\tau} P'$. Per induzione, $P_1 \mid !P_1 \longmapsto P'$; per (R-Struct) e (S-Rep), $!P_1 \longmapsto P'$.

(LTS-Eq): simile a (LTS-Rep). ∎

Proposizione 10 *Se* $P \longmapsto P'$, *allora* $P \xrightarrow{\tau} \equiv P'$.

Dim.: per induzione sulla lunghezza della più breve inferenza per $P \longmapsto P'$.

Passo Base: l'unico assioma è (R-Com). Quindi, per definizione, $P \triangleq a(x).P_1 \mid \overline{a}\langle b\rangle.P_2$ e $P' \triangleq P_1[b/x] \mid P_2$. Allora, usando (LTS-In), (LTS-Out) e (LTS-Com), abbiamo che

$$\dfrac{a.(x).P_1 \xrightarrow{ab} P_1[b/x] \qquad \overline{a}\langle b\rangle.P_2 \xrightarrow{\bar{a}b} P_2}{P \xrightarrow{\tau} P'}$$

Passo Induttivo: distinguiamo l'ultima regola usata.

(R-PAR)**:** in questo caso, $P \triangleq P_1 \mid P_2$, $P_1 \longmapsto P_1'$ e $P' \triangleq P_1' \mid P_2$. Per induzione, $P_1 \xrightarrow{\tau} \equiv P_1'$, cioè $P_1 \xrightarrow{\tau} P_1''$ e $P_1'' \equiv P_1'$. Per (LTS-PAR), $P_1 \mid P_2 \xrightarrow{\tau} P_1'' \mid P_2$ e $P_1'' \mid P_2 \equiv P_1' \mid P_2$.

(R-RES)**:** in questo caso, $P \triangleq (\nu n)P_1$, $P' \triangleq (\nu n)P_1'$ e $P_1 \longmapsto P_1'$. Per induzione, $P_1 \xrightarrow{\tau} P_1''$ e $P_1'' \equiv P_1'$. Per (LTS-RES), $(\nu n)P_1 \xrightarrow{\tau} (\nu n)P_1''$ e $(\nu n)P_1'' \equiv (\nu n)P_1'$.

(R-STRUCT)**:** in questo caso, $P \equiv Q$, $Q \longmapsto Q'$ e $Q' \equiv P'$. Per induzione $Q \xrightarrow{\tau} \equiv Q'$. A questo punto è necessaria una seconda induzione per poter dimostrare questo caso. Lavoreremo sulla profondità della derivazione di $P \equiv Q$. Inoltre assumiamo che la prima regola usata per inferire $Q \longmapsto Q'$ non sia (R-STRUCT), altrimenti l'inferenza di $P \longmapsto P'$ potrebbe essere accorciata.

Passo Base, 2° induzione: abbiamo 14 sotto-casi: uno per ogni assioma di Tabella 1.1, tranne (S-ABS) (non applicabile poichè $\mathbf{0} \not\longmapsto$); in qualche caso considereremo anche la versione simmetrica dell'assioma, in modo da poter ignorare la regola (S-SIMM) nel passo induttivo.

(S-REFL)**:** banale.

(S-CONV)**:** basta ridenominare (se necessario) il nome legato in un bound output (ricordiamo che l'LTS è definito a meno di alfa-conversione).

(S-ID)**:** in questo caso, $P \triangleq R \mid \mathbf{0}$ e $Q \triangleq R$. Per la prima induzione, $R \xrightarrow{\tau} \equiv Q'$, da cui, per (LTS-PAR), $R \mid \mathbf{0} \xrightarrow{\tau} \equiv Q' \mid \mathbf{0} \equiv Q'$.

Simmetrico di (S-ID)**:** banale.

(S-COM)**:** in questo caso, $P \triangleq P_1 \mid P_2$ e $Q \triangleq P_2 \mid P_1$. La tesi segue dalla prima induzione e dalla doppia formulazione delle regole (LTS-PAR), (LTS-COM) e (LTS-CLOSE).

(S-ASS)**:** in questo caso, $P \triangleq P_1 \mid (P_2 \mid P_3)$ e $Q \triangleq (P_1 \mid P_2) \mid P_3$. Per il Lemma 8(4), ci sono sei possibilità per $Q \xrightarrow{\tau} \equiv Q'$:

1. $P_1 \mid P_2 \xrightarrow{\tau} Q_1$ e $Q' \equiv Q_1 \mid P_3$
2. $P_3 \xrightarrow{\tau} P_3'$ e $Q' \equiv (P_1 \mid P_2) \mid P_3'$
3. $P_1 \mid P_2 \xrightarrow{ab} Q_1$, $P_3 \xrightarrow{\bar{a}b} P_3'$ e $Q' \equiv Q_1 \mid P_3'$
4. $P_1 \mid P_2 \xrightarrow{\bar{a}b} Q_1$, $P_3 \xrightarrow{ab} P_3'$ e $Q' \equiv Q_1 \mid P_3'$
5. $P_1 \mid P_2 \xrightarrow{ab} Q_1$, $P_3 \xrightarrow{\bar{a}(b)} P_3'$ e $Q' \equiv (\nu b)(Q_1 \mid P_3')$

6. $P_1 \mid P_2 \xrightarrow{\bar{a}(b)} Q_1$, $P_3 \xrightarrow{ab} P_3'$ e $Q' \equiv (\nu b)(Q_1 \mid P_3')$

Analizzeremo solo il caso (6) in quanto gli altri sono simili o più semplici. Il caso (6) porta a due sottocasi:

- $P_1 \xrightarrow{\bar{a}(b)} P_1'$, con $b \notin fn(P_2)$, e $Q_1 \triangleq P_1' \mid P_2$. Usando (LTS-Par) e (LTS-Close), abbiamo la seguente inferenza:

$$\frac{P_1 \xrightarrow{\bar{a}(b)} P_1' \qquad \dfrac{P_3 \xrightarrow{ab} P_3'}{P_2 \mid P_3 \xrightarrow{ab} P_2 \mid P_3'}}{P_1 \mid (P_2 \mid P_3) \xrightarrow{\tau} (\nu b)(P_1' \mid (P_2 \mid P_3'))}$$

  Ora $(\nu b)(P_1' \mid (P_2 \mid P_3')) \equiv (\nu b)((P_1' \mid P_2) \mid P_3') \equiv (\nu b)(Q_1 \mid P_3') \equiv Q' \equiv P'$.

- $P_2 \xrightarrow{\bar{a}(b)} P_2'$, con $b \notin fn(P_1)$, e $Q_1 \triangleq P_1 \mid P_2'$. Similmente

$$\frac{\dfrac{P_2 \xrightarrow{\bar{a}(b)} P_2' \qquad P_3 \xrightarrow{ab} P_3'}{P_2 \mid P_3 \xrightarrow{\tau} (\nu b)(P_2' \mid P_3')}}{P_1 \mid (P_2 \mid P_3) \xrightarrow{\tau} P_1 \mid (\nu b)(P_2' \mid P_3')}$$

  e $P_1 \mid (\nu b)(P_2' \mid P_3') \equiv P'$.

Simmetrico di (S-Ass)**:** simile.

(S-Eq)**:** in questo caso, $P \triangleq [a = b]P_1$ e $Q \triangleq P_1$. Per induzione, $P_1 \xrightarrow{\tau} \equiv Q'$ da cui, per (LTS-Eq), $[a = a]P_1 \xrightarrow{\tau} \equiv Q'$.

Simmetrico di (S-Eq)**:** banale.

(S-Rep)**:** simile al caso precedente, ma usa (LTS-Rep).

Simmetrico di (S-Rep)**:** banale.

(S-RCom)**:** banale.

(S-Ext)**:** in questo caso, $P \triangleq P_1 \mid (\nu b)P_2$ e $Q \triangleq (\nu b)(P_1 \mid P_2)$, con $b \notin fn(P_1)$. Per il Lemma 8(4), ci sono sei possibilità per $(\nu b)(P_1 \mid P_2) \xrightarrow{\tau} \equiv Q'$:

1. $P_1 \xrightarrow{\tau} Q_1$ e $Q' \equiv (\nu b)(Q_1 \mid P_2)$
2. $P_2 \xrightarrow{\tau} Q_1$ e $Q' \equiv (\nu b)(P_1 \mid Q_1)$
3. $P_1 \xrightarrow{ac} P_1'$, $P_2 \xrightarrow{\bar{a}c} P_2'$ e $Q' \equiv (\nu b)(P_1' \mid P_2')$
4. $P_1 \xrightarrow{\bar{a}c} P_1'$, $P_2 \xrightarrow{ac} P_2'$ e $Q' \equiv (\nu b)(P_1' \mid P_2')$
5. $P_1 \xrightarrow{ac} P_1'$, $P_2 \xrightarrow{\bar{a}(c)} P_2'$ e $Q' \equiv (\nu b)(\nu c)(P_1' \mid P_2')$

6. $P_1 \xrightarrow{\bar{a}(c)} P_1'$, $P_2 \xrightarrow{ac} P_2'$ e $Q' \equiv (\nu b)(\nu c)(P_1' \mid P_2')$

Analizziamo solo il caso (3) che porta ai seguenti sottocasi:

- $c \neq b$; inoltre sappiamo che $b \neq a$ poichè $b \notin fn(P_1)$ mentre $a \in fn(P_1)$. Allora, per (LTS-Res) e (LTS-Com), abbiamo che

$$\frac{P_1 \xrightarrow{ac} P_1' \qquad \dfrac{P_2 \xrightarrow{\bar{a}c} P_2'}{(\nu b)P_2 \xrightarrow{\bar{a}c} (\nu b)P_2'}}{P_1 \mid (\nu b)P_2 \xrightarrow{\tau} P_1' \mid (\nu b)P_2'}$$

  e $P_1' \mid (\nu b)P_2' \equiv (\nu b)(P_1' \mid P_2')$. Questo è possibile poichè $fn(P_1') = fn(P_1) \cup \{c\}$ e $b \notin fn(P_1) \cup \{c\}$. Da ciò segue banalmente che $P \xrightarrow{\tau} \equiv P'$.

- $c = b$. In tal caso, usando (LTS-Open) e (LTS-Close), si ha che

$$\frac{P_1 \xrightarrow{ab} P_1' \qquad \dfrac{P_2 \xrightarrow{\bar{a}b} P_2'}{(\nu b)P_2 \xrightarrow{\bar{a}(b)} P_2'}}{P_1 \mid (\nu b)P_2 \xrightarrow{\tau} (\nu b)(P_1' \mid P_2')}$$

  da cui è facile concludere.

Simmetrico di (S-Ext): simile al caso precedente, ma più semplice.

Passo induttivo, 2° induzione: distinguiamo in base all'ultima regola usata:

(S-Trans): in questo caso, $P \equiv P_1$, $P_1 \equiv Q$ e $Q \xrightarrow{\tau} \equiv Q'$. Per la seconda induzione applicata a $P_1 \equiv Q$ e $Q \xrightarrow{\tau} \equiv Q'$, abbiamo che $P_1 \xrightarrow{\tau} \equiv Q'$. Di nuovo per la seconda induzione, questa volta applicata a $P \equiv P'$ e $P_1 \xrightarrow{\tau} \equiv Q'$, abbiamo che $P \xrightarrow{\tau} \equiv Q' \equiv P'$.

(S-Res): in questo caso, $P \triangleq (\nu n)P_1$, $Q \triangleq (\nu n)P_2$ e $P_1 \equiv P_2$. Poichè la prima regola per inferire $Q \longmapsto Q'$ non è (R-Struct), è facile convincersi che $Q' \equiv (\nu n)Q''$ e $P_2 \longmapsto Q''$. Per la prima induzione si ha che $P_2 \xrightarrow{\tau} \equiv Q''$. Per la seconda induzione, $P_1 \xrightarrow{\tau} \equiv Q''$ da cui, usando (LTS-Res), si ha che $P \triangleq (\nu n)P_1 \xrightarrow{\tau} \equiv (\nu n)Q'' \equiv Q' \equiv P'$.

(S-Par): simile al caso precedente.

∎

1.5 Bisimulazione nel π-calcolo

Uno dei vantaggi dell'LTS sta nella possibilità di costruire su di esso una bisimulazione simile a quella del CCS.

Definizione 11 *Una relazione simmetrica $\mathcal{R}$ tra processi è una* bisimulazione forte *se, per ogni $P\mathcal{R}Q$ e per ogni $P \xrightarrow{\alpha} P'$ con $bn(\alpha) \cap fn(P,Q) = \emptyset$, si ha che esiste Q' tale che $Q \xrightarrow{\alpha} Q'$ e $P'\mathcal{R}Q'$. La bisimilarità, $\sim$, è la più grande bisimulazione.*

Tale definizione può essere spiegata intuitivamente in termini di un *bisimulation game*. Abbiamo due partecipanti P e Q. P sfida Q compiendo un'azione α e riducendosi ad un qualche P'. A questo punto, Q deve essere in grado di rispondere alla sfida compiendo anch'esso α e riducendosi ad un Q' bisimile a P'. Se ciò non è possibile, P vince il gioco e i due processi di partenza non sono bisimili. Se il gioco finisce "in parità" per ogni sfida possibile, P e Q sono bisimili.

Da notare è il fatto che la definizione di bisimilarità differisce da quella del CCS solo per la condizione $bn(\alpha) \cap fn(P,Q) = \emptyset$. Ciò è dovuto alla gestione dei nomi ristretti. Si osservi questo esempio:

$$P \triangleq (\nu a)\overline{b}\langle a\rangle \qquad Q \triangleq (\nu a)\overline{b}\langle a\rangle \mid (\nu w)\overline{w}\langle a\rangle$$

Il processo P vuole spedire il nome ristretto a su b, mentre il processo Q fa la stessa cosa e in parallelo manda sul canale ristretto w il nome libero a. Questi due processi sono equivalenti perchè, essendo w un canale ristretto, il processo $(\nu w)\overline{w}\langle a\rangle$ non genera nessun comportamento. Ora consideriamo la sfida $P \xrightarrow{\overline{b}(a)} \mathbf{0}$; Q non riesce a rispondere poichè la side condition di (LTS-Par) rende la regola non applicabile, perchè $a = bn(\overline{b}(a)) \in fn((\nu w)\overline{w}\langle a\rangle)$. Per evitare questa situazione, nel bisimulation game la sfida $\overline{b}(a)$ non è consentita poichè $bn(\overline{b}(a)) \cap fn(P,Q) \neq \emptyset$; ridenominando i nomi bound di P e Q con nomi fresh (per esempio ridenominiamo a con c), si ottiene che:

$$(\nu c)\overline{b}\langle c\rangle \quad \sim \quad (\nu c)\overline{b}\langle c\rangle \mid (\nu w)\overline{w}\langle a\rangle$$

Andiamo ora a dimostrare che la bisimulazione è un'equivalenza "interessante". In CCS [10] questo è testimoniato dal fatto che la bisimulazione tiene conto della ramificazione dei processi, al contrario dell'equivalenza di linguaggi (equivalenza a tracce). Un argomento simile vale anche per il π-calcolo [3]. Tuttavia, noi seguiremo qui un approccio alternativo che consiste nel dimostrare che è una congruenza, cioè è un'equivalenza chiusa per contesti.

Definizione 12 *$\sim$ è una congruenza rispetto ad un operatore $op(\cdot)$ se $P \sim Q$ implica $op(P) \sim op(Q)$.*

Il fatto che una relazione sia una congruenza la rende usabile in maniera semplice perchè permette di fare ragionamenti equazionali e composizionali. Infatti, per dimostrare l'equivalenza tra due processi, basterà dimostrare l'equivalenza tra sottoprocessi opportuni.

Nel caso del π-calcolo si hanno sei operatori. Dimostrare la congruenza per gli operatori di matching ed output è banale.

Proposizione 13 *Se $P \sim Q$, allora $[a = b]P \sim [a = b]Q$.*

Dim.: Se $a \neq b$ il caso è banale, poichè sia $[a = b]P$ che $[a = b]Q$ sono equivalenti a $\mathbf{0}$. Altrimenti, è facile dimostrare che $[a = a]P \sim P$ e $[a = a]Q \sim Q$; la tesi segue per transitività di $\sim$.

La dimostrazione per l'operatore di output è simile. ∎

La prossima proposizione mostra che $\sim$ è una congruenza rispetto sia alla restrizione che e al parallelo. Infatti, per $\widetilde{m} = \emptyset$ abbiamo la congruenza rispetto al parallelo e per $P_3 = \mathbf{0}$ abbiamo la congruenza rispetto alla restrizione.

Proposizione 14 *Se $P_1 \sim P_2$, allora per ogni P_3 e per ogni $\widetilde{m}$ si ha che $(\nu\widetilde{m})(P_1|P_3) \sim (\nu\widetilde{m})(P_2|P_3)$.*

Dim.: basta dimostrare che

$$\mathcal{R} \triangleq \{(\nu\widetilde{n})(P|R), (\nu\widetilde{n})(Q|R) : P \sim Q \ \wedge \ bn(R) \cap fn(P, Q) = \emptyset\}$$

è una bisimulazione. Partiamo da $(\nu\widetilde{n})(P|R) \xrightarrow{\alpha} P'$; per definizione dell'LTS, si deve avere:

$$\dfrac{\dfrac{}{P|R \xrightarrow{\alpha'} P''}}{\dfrac{\cdots \qquad \cdots}{(\nu\widetilde{n})(P|R) \xrightarrow{\alpha} P'}}$$

La derivazione sfrutta una serie di applicazioni della regola (LTS-Res) ed al massimo una volta la (LTS-Open) se $\alpha \triangleq \overline{a}(b)$ per $b \in \widetilde{n}$; in tal caso, $\alpha' = \overline{a}b$ e $P' \triangleq (\nu\widetilde{n'})P''$ con $\widetilde{n'} \triangleq \widetilde{n} - \{b\}$; altrimenti $\alpha' = \alpha$ e $P' \triangleq (\nu\widetilde{n})P''$. Per il Lemma 8(4) ho sei casi:

$P \xrightarrow{\alpha'} P_1$ **e** $P'' \triangleq P_1|R$**:** per ipotesi di bisimulazione, $Q \xrightarrow{\alpha'} Q_1$ e $P_1 \sim Q_1$. Allora, mimando la derivazione di sopra, si ottiene che $(\nu\widetilde{n})(Q|R) \xrightarrow{\alpha} Q'$ e $P'\mathcal{R}Q'$. Infatti, se si è usata la (LTS-Open), $Q' \triangleq (\nu\widetilde{n'})(Q_1|R)$ e quindi $P' \triangleq (\nu\widetilde{n'})(P_1|R)\ \mathcal{R}\ (\nu\widetilde{n'})(Q_1|R) \triangleq Q'$; altrimenti, $Q' \triangleq (\nu\widetilde{n})(Q_1|R)$ ed il discorso è analogo.

$R \xrightarrow{\alpha} R'$ **e** $P'' \triangleq P|R'$**:** simile al caso precedente.

$P \xrightarrow{ab} P_1$**,** $R \xrightarrow{\bar{a}b} R'$ **e** $P'' \triangleq P_1|R'$**:** per ipotesi di bisimulazione, $Q \xrightarrow{ab} Q_1$ e $P_1 \sim Q_1$. Allora, mimando la derivazione, abbiamo che $(\nu\widetilde{n})(Q|R) \xrightarrow{\tau} Q'$, dove $Q' \triangleq (\nu\widetilde{n})(Q_1|R')$; infatti, poichè $\alpha = \tau$, la (LTS-OPEN) non può essere stata applicata. Per definizione di $\mathcal{R}$, si ha quindi che $P'\mathcal{R}Q'$.

$P \xrightarrow{\bar{a}b} P_1$**,** $R \xrightarrow{ab} R'$ **e** $P'' \triangleq P_1|R'$**:** simile al caso precedente.

$P \xrightarrow{ab} P_1$**,** $R \xrightarrow{\bar{a}(b)} R'$**,** $b \notin fn(P)$ **e** $P'' \triangleq (\nu b)(P_1|R')$**:** per ipotesi di bisimulazione, $Q \xrightarrow{ab} Q_1$ e $P_1 \sim Q_1$. Per (LTS-CLOSE), $Q|R \xrightarrow{\tau} (\nu b)(Q_1|R')$ e, per (LTS-RES), $(\nu\widetilde{n})(Q|R) \xrightarrow{\tau} (\nu\widetilde{n})(\nu b)(Q_1|R') \triangleq Q'$. Per definizione di $\mathcal{R}$, $P'\mathcal{R}Q'$.

$P \xrightarrow{\bar{a}(b)} P_1$**,** $R \xrightarrow{ab} R'$**,** $b \notin fn(R)$ **e** $P'' \triangleq (\nu b)(P_1|R')$**:** simile al caso precedente. ∎

Proposizione 15 *Se* $P \sim Q$*, allora* $!P \sim !Q$*.*

Dim.: Anzitutto, abbreviamo la composizione parallela di n copie di P come P^n; chiaramente, $P^0 = \mathbf{0}$. Applicando la Proposizione 14 $n-1$ volte, si ottiene che $P \sim Q$ implica $P^n \sim Q^n$.

Ora, definiamo l'approssimazione n-esima di $\sim$ come segue:

$$\begin{array}{ll} P \sim^0 Q & \text{sempre} \\ P \sim^{k+1} Q & \text{sse } \forall P \xrightarrow{\alpha} P'\ \exists Q'.(Q \xrightarrow{\alpha} Q' \wedge P' \sim^k Q'), \text{ e viceversa} \end{array}$$

La proprietà fondamentale della famiglia $\{\sim^n\}_{n\geq 0}$ è la seguente:

$$P \sim Q \text{ se e solo se } P \sim^n Q, \text{ per ogni } n.$$

L'implicazione '$\Rightarrow$' è per induzione su n e non presenta particolari problemi; l'implicazione '$\Leftarrow$' si ottiene dimostrando che $\bigcap_n \sim^n$ è una bisimulazione (intuitivamente, tale fatto è abbastanza chiaro; per una dimostrazione formale si rimanda a [16]).

Dimostriamo ora, per induzione su n, che $R^k \sim^n !R$ per ogni $k \geq 2n$. Il caso base è banale, poichè $\sim^0$ eguaglia tutti i processi. Per il passo induttivo, sia $R^k \xrightarrow{\alpha} R'$ (il caso in cui la sfida venga da $!R$ è simile); si hanno due casi:

1. $R \xrightarrow{\alpha} R''$ and $R' \equiv R'' \mid R^{k-1}$, oppure
2. $R \mid R \xrightarrow{\tau} R''$ and $R' \equiv R'' \mid R^{k-2}$.

Nel primo caso, $!R \xrightarrow{\alpha} R'' \mid !R$ e $R'' \mid R^{k-1} \sim^{n-1} R'' \mid !R$: ciò segue dall'ipotesi induttiva $R^{k-1} \sim^{n-1} !R$ e dalla congruenza rispetto al parallelo di $\sim^{n-1}$ (che può essere dimostrata similmente alla Proposizione 14). Nel secondo caso, $!R \xrightarrow{\alpha} R'' \mid !R$ e, sempre per induzione e chiusura per parallelo, $R'' \mid R^{k-2} \sim^{n-1} R'' \mid !R$.

Quindi, comunque fissiamo n, abbiamo che $P \sim Q$ implica $P^{2n} \sim Q^{2n}$; pertanto, $!P \sim^n P^{2n} \sim^n Q^{2n} \sim^n !Q$. Per transitività di $\sim^n$ (banalmente dimostrabile), abbiamo $!P \sim^n !Q$; data l'arbitrarietà di n, ciò vale per ogni approssimazione di $\sim$ e, quindi, $!P \sim !Q$. ∎

Resta solo da vedere se la bisimulazione è una congruenza rispetto al prefisso di input. Il seguente esempio mostra che ciò *non* è vero. Se $x \neq a$ allora banalmente

$$[x = a]\overline{x}\langle b\rangle \sim \mathbf{0}$$

poichè l'output $\overline{x}\langle b\rangle$ non viene mai eseguito. Ma se si aggiunge un prefisso di input questo non è più vero:

$$c(x).[x = a]\overline{x}\langle b\rangle \not\sim c(x).\mathbf{0}$$

Infatti il processo di sinistra può ricevere a dal canale c e poi eseguire l'output:

$$c(x).[x = a]\overline{x}\langle b\rangle \xrightarrow{ca} [a = a]\overline{a}\langle b\rangle \xrightarrow{\bar{a}b} \mathbf{0}$$

Nel processo di destra questo non è possibile e la congruenza rispetto all'input non vale.

Andiamo quindi a definire una sottorelazione di $\sim$ che sia una congruenza rispetto a tutti gli operatori e che sia la più grande tra quelle che rispettano questa proprietà. A tale scopo, indichiamo con σ una sostituzione di nomi a nomi, cioè una funzione parziale a dominio finito da $\mathcal{N}$ a $\mathcal{N}$. Indichiamo inoltre con $P\sigma$ l'applicazione di σ a tutti i nomi liberi di P (assumiamo che σ agisca come l'identità su nomi che non appertengono al suo dominio), con un'eventuale alfa-conversione di nomi bound di P che appaiono nel codominio di σ.

Definizione 16 *$\sim^c$ è la più grande relazione tale che $P \sim^c Q$ se e solo se per ogni sostituzione σ abbiamo che $P\sigma \sim Q\sigma$.*

Teorema 17 *$\sim^c$ è la più grande congruenza contenuta in $\sim$.*

Dim.: Si deve dimostrare che $\sim^c \subseteq \sim$, che $\sim^c$ è una congruenza e che è la più grande. Procediamo con le dimostrazioni di queste tre affermazioni.

1. $P \sim^c Q$ implica banalmente che $P \sim Q$: basta utilizzare la sostituzione vuota.

2. Il caso cruciale è la congruenza rispetto al prefisso di input. Partendo da $P \sim^c Q$, si deve dimostrare che $a(x).P \sim^c a(x).Q$, cioè che, per ogni σ, si ha $(a(x).P)\sigma \sim (a(x).Q)\sigma$. Distinguiamo due casi:

   - $a \notin dom(\sigma)$. In questo caso, $(a(x).P)\sigma \triangleq a(x).(P\sigma)$ e $(a(x).Q)\sigma \triangleq a(x).(Q\sigma)$. Consideriamo l'unica sfida che $(a(x).P)\sigma$ può fare (il viceversa è analogo): $a(x).(P\sigma) \xrightarrow{ab} (P\sigma)[b/x]$. Banalmente, $(P\sigma)[b/x] \triangleq P([b/x] \circ \sigma)$; similmente, $a(x).(Q\sigma) \xrightarrow{ab} Q([b/x] \circ \sigma)$. Ora, da $P \sim^c Q$ segue che $P([b/x] \circ \sigma) \sim Q([b/x] \circ \sigma)$. Questo basta a dimostrare che $a(x).P \sim^c a(x).Q$.
   - $a \in dom(\sigma)$. Sia $\sigma(a) = c$, la dimostrazione è analoga al caso precedente solo che qui avremo $(a(x).P)\sigma \triangleq c(x).(P\sigma)$ e $(a(x).Q)\sigma \triangleq c(x).(Q\sigma)$.

   Per i restanti operatori, la dimostrazione segue dal fatto che $\sim$ è una congruenza rispetto ad essi. Prendiamo il parallelo come caso rappresentativo; gli altri sono analoghi. Si deve dimostrare che $P|R \sim^c Q|R$. Per definizione, $P\sigma \sim Q\sigma$ per ogni σ. In quanto la bisimulazione è congruenza rispetto al parallelo, si ha che $P\sigma \mid R\sigma \sim Q\sigma \mid R\sigma$, cioè $(P|R)\sigma \sim (Q|R)\sigma$.

3. Dimostriamo che, se $\sim'$ è una congruenza contenuta in $\sim$, allora $\sim' \subseteq \sim^c$. Sia $P \sim' Q$; si deve dimostrare che $P\sigma \sim Q\sigma$, per ogni σ. Fissiamo arbitrariamente $\sigma \triangleq [b_1, ..., b_n/x_1, ..., x_n]$; poichè $\sim'$ è una congruenza, abbiamo che

$$\begin{aligned}(\nu a)(a(x_1).\cdots.a(x_n).P \mid \overline{a}\langle b_1\rangle.\cdots.\overline{a}\langle b_n\rangle) \\ \sim' \quad (\nu a)(a(x_1).\cdots.a(x_n).Q \mid \overline{a}\langle b_1\rangle.\cdots.\overline{a}\langle b_n\rangle)\end{aligned}$$

   per a fresh. Inoltre, poichè $\sim'$ è una bisimulazione, dopo le n comunicazioni avremo che:

$$P[b_1, ..., b_n/x_1, ..., x_n] \;\sim'\; Q[b_1, ..., b_n/x_1, ..., x_n]$$

   cioè $P\sigma \sim' Q\sigma$. Data l'arbitrarietà di σ e il fatto che $\sim' \subseteq \sim$, ciò dimostra $P \sim^c Q$. ∎

Concludiamo questo capitolo mostrando la definizione di bisimulazione debole per il π-calcolo. Non commentiamo tale definizione nè diamo alcun

risultato per essa, visto che tutto ciò che abbiamo provato in questo capitolo per la bisimulazione forte può essere adattato alla bisimulazione debole. Scriviamo $\Rightarrow$ per intendere la chiusura transitiva e riflessiva di $\xrightarrow{\tau}$, cioè per indicare zero o più τ-transizioni. Scriviamo inoltre $\stackrel{\alpha}{\Longrightarrow}$ per indicare $\Rightarrow \xrightarrow{\alpha} \Rightarrow$; infine, $\stackrel{\hat{\alpha}}{\Longrightarrow}$ denota $\stackrel{\alpha}{\Longrightarrow}$, se $\alpha \neq \tau$, e $\Rightarrow$ altrimenti.

Definizione 18 *Una relazione simmetrica $\mathcal{R}$ tra processi è una* bisimulazione debole *se, per ogni $P\mathcal{R}Q$ e per ogni $P \xrightarrow{\alpha} P'$ con $bn(\alpha) \cap fn(P,Q) = \emptyset$, si ha che esiste Q' tale che $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ e $P'\mathcal{R}Q'$. La* bisimilarità debole*, $\approx$, è la più grande bisimulazione debole.*

Capitolo 2

Varianti del π-calcolo

2.1 Il π-calcolo Poliadico

La prima estensione del π-calcolo presentato nel Capitolo 1 (che qui chiameremo *monadico* perchè la comunicazione ha per oggetto un singolo canale) consiste nel permettere il passaggio di tuple di canali, anzichè di canali singoli. Questa estensione del π-calcolo viene definita π-calcolo *poliadico* ed è stata introdotta in [11].

2.1.1 Passaggio di Tuple di Canali nella Comunicazione

La sintassi del π-calcolo poliadico è

$$P \quad ::= \quad \mathbf{0} \mid a(\widetilde{x}).P \mid \overline{a}\langle\widetilde{b}\rangle.P \mid (\nu a)P \mid P_1|P_2 \mid\, !P \mid\ \ [a=b]P$$

dove $\widetilde{m}$ è una qualsiasi sequenza di nomi. Si noti che il π-calcolo monadico si ottiene prendendo sempre sequenze di lunghezza 1, mentre il CCS si ottiene prendendo sempre sequenze di lunghezza 0.

La semantica operazionale si ottiene invece modificando la regola di comunicazione in maniera ovvia:

$$\text{(R-Com)} \quad a(\widetilde{x}).P \mid \overline{a}\langle\widetilde{b}\rangle.Q \longmapsto P[\widetilde{b}/\widetilde{x}] \mid Q$$

dove $[\widetilde{b}/\widetilde{x}]$ denota la sostituzione, componente per componente, dell'i-esimo elemento di $\widetilde{x}$ con l'i-esimo elemento di $\widetilde{b}$.

Esempio 19 Come già accennato nell'esempio 7, la seconda implementazione dei booleani può essere resa più efficiente passando i due nomi usati per il

test simultaneamente. Con il π-calcolo poliadico, questo è possibile. Pertanto, abbiamo che

$$\begin{array}{rcl} True_a & \triangleq & a(y,z).\overline{y}\langle y\rangle \\ False_a & \triangleq & a(y,z).\overline{z}\langle z\rangle \\ Test_a & \triangleq & (\nu c,d)\overline{a}\langle c,d\rangle.(c(x).P \mid d(x).Q) \end{array}$$

In tal caso, il processo di testing richiede solo due riduzioni

$$Test_a(P;Q) \mid True_a \longmapsto\longmapsto (\nu c,d)(P \mid d(x).Q) \approx P$$

2.1.2 Un Type System per il π-calcolo Poliadico

Un problema tipico nel π-calcolo poliadico sorge quando nella (R-Com) $\widetilde{x}$ e $\widetilde{b}$ hanno lunghezze diverse. Il modo più naturale di vedere tale problema è quello di considerarlo come un errore di programmazione. Ciò è motivabile con la seguente analogia tra il π-calcolo ed il linguaggio di programmazione C.

π-calcolo	C
Input	Definizione di funzione
Variabili	Parametri formali
Output	Invocazione di funzione
Argomenti	Parametri attuali
Comunicazione	Attivazione di una chiamata di funzione

Quindi, per analogia, la comunicazione tra input ed output di diversa arità corrisponde al chiamare una funzione con un numero errato di parametri. Come nel linguaggio di programmazione C il compilatore rileva un errore di tipo, così vogliamo sviluppare per il π-calcolo poliadico un sistema di tipi che rilevi tali anomalie. Come prima cosa definiamo formalmente le anomalie che vogliamo evitare. A tal scopo, in Tabella 2.1 definiamo il predicato $P\uparrow$. Intuitivamente $P\uparrow$ va letto come: "In P c'è un errore di comunicazione dovuto alla diversa lunghezza dei parametri coinvolti".

Per rilevare staticamente errori runtime, associamo ad ogni canale un tipo che descrive le comunicazioni lecite su quel canale. Quindi definiamo un ambiente di tipaggio Γ come una funzione parziale dall'insieme dei nomi $\mathcal{N}$ all'insieme dei tipi $\mathcal{T}$. Ma come dobbiamo definire $\mathcal{T}$? Il modo più semplice sarebbe quello di scegliere $\mathcal{T} = \mathbb{N}$, in cui $\Gamma(a)$ esprime il numero di valori che possono/devono essere passati in una comunicazione sul canale a.

$$\text{(E-Com)}\ \frac{}{a(\widetilde{x}).P \mid \overline{a}\langle\widetilde{b}\rangle.Q \uparrow}\ {\scriptstyle |\widetilde{x}|\neq|\widetilde{b}|} \qquad \text{(E-Par)}\ \frac{P \uparrow}{P \mid Q \uparrow}$$

$$\text{(E-Res)}\ \frac{P \uparrow}{(\nu a)P \uparrow} \qquad \text{(E-Struct)}\ \frac{P \uparrow \quad P \equiv Q}{Q \uparrow}$$

Tabella 2.1: Errore di arità in una comunicazione

Consideriamo ora le seguenti (intuitive) regole di tipaggio:

$$\frac{\Gamma(a) = |\widetilde{b}| \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}\langle\widetilde{b}\rangle.P} \qquad \frac{\Gamma(a) = |\widetilde{x}| \qquad \Gamma \vdash P}{\Gamma \vdash a(\widetilde{x}).P} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$$

e proviamo ad applicare il sistema di tipi risultante a due esempi.

Esempio 20 Consideriamo il processo poliadico

$$\overline{a}\langle b, c\rangle.P \mid a(x).Q$$

Per tipare $\overline{a}\langle b, c\rangle.P$ deve essere $\Gamma(a) = 2$, mentre per tipare $a(x).Q$ deve essere $\Gamma(a) = 1$; questo è in contraddizione con il fatto che Γ sia una funzione. Quindi il termine non è tipabile, come ci aspettavamo.

Esempio 21 Consideriamo

$$a(x).x(z) \mid \overline{a}\langle c\rangle.\overline{c}\langle b, b\rangle$$

Prendendo Γ tale che

$$\Gamma(a) = 1 \qquad \Gamma(x) = 1 \qquad \Gamma(c) = 2 \qquad \Gamma(z) = \ldots \qquad \Gamma(b) = \ldots$$

il termine è ben-tipato con questo sistema dei tipi. Ma con una sola comunicazione otteniamo

$$a(x).x(z) \mid \overline{a}\langle c\rangle.\overline{c}\langle b, b\rangle \longmapsto c(z) \mid \overline{c}\langle b, b\rangle$$

cioè un termine tipabile ha generato un termine non tipabile. Vorremmo un sistema di tipi che accetti solo processi che non generino mai errori.

Il problema mostrato dall'Esempio 21 è dovuto al fatto che non è sufficiente tener traccia solo del numero di valori passati (cioè $\mathcal{T} = \mathbb{N}$), ma è necessario tener traccia anche del tipo di ogni valore (come anche avviene nei

$$\text{(T-Nil)}\quad \frac{}{\Gamma \vdash \mathbf{0}} \qquad \text{(T-Match)}\quad \frac{\Gamma \vdash P}{\Gamma \vdash [a=b]P} \qquad \text{(T-Rep)}\quad \frac{\Gamma \vdash P}{\Gamma \vdash\, !P}$$

$$\text{(T-Par)}\quad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \text{(T-Res)}\quad \frac{\exists T.\Gamma, a:T \vdash P}{\Gamma \vdash (\nu a)P}\ a \notin dom(\Gamma)$$

$$\text{(T-Out)}\quad \frac{\Gamma(a) = [T_1,\ldots,T_k] \qquad \forall i = 1,\ldots,k.\Gamma(b_i) = T_i \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}\langle b_1,\ldots,b_k\rangle.P}$$

$$\text{(T-In)}\quad \frac{\Gamma(a) = [T_1,\ldots,T_k] \quad \Gamma, x_1:T_1,\ldots,x_k:T_k \vdash P}{\Gamma \vdash a(x_1,\ldots,x_k).P}\ \{x_1,\ldots,x_k\} \cap dom(\Gamma) = \emptyset$$

Tabella 2.2: Regole del Type System

linguaggi tipo il C). Per ovviare a ciò, diamo una definizione di tipo ricorsiva che ci permetta di tipare ogni termine. I tipi di cui abbiamo bisogno sono definiti ricorsivamente come segue:

$$T \quad ::= \quad [T_1,\ldots,T_k] \qquad k \geq 0$$

Rappresentiamo un ambiente di tipaggio Γ come una sequenza di associazioni nome-tipo $n_1 : T_1, \ldots, n_k : T_k$ tale che $n_i \neq n_j$ se $i \neq j$. Chiaramente, $dom(\Gamma) \triangleq \{n_1,\ldots,n_k\}$.

Le regole del sistema di tipi sono definite in Tabella 2.2. La regola (T-Nil) dice che $\mathbf{0}$ è sempre tipabile; le regole (T-Match) e (T-Rep) dicono che, per tipare $[a=b]P$ e $!P$, basta essere in grado di tipare P; (T-Par) dice che la composizione parallela di due processi è tipabile solo se i due sottoprocessi lo sono; (T-Res) dice che, per tipare $(\nu a)P$, basta trovare un opportuno tipo T da assegnare al nuovo nome; l'uso che P farà di a è regolato da tale tipo. Le regole cruciali del type system sono (T-In) e (T-Out). Esse dicono che, quando un processo usa un certo canale, il numero di messaggi inviati/ricevuti da esso è quello stabilito dal tipo associato al canale. Inoltre, nel caso dell'output, bisogna anche verificare che i dati spediti siano del tipo prescritto dal canale, mentre nel caso dell'input bisogna anche verificare che la continuazione sia tipabile assumendo che le variabili di input abbiano il tipo specificato nel tipo di a.

Rivediamo ora gli Esempi 20 e 21 nell'ambito del nuovo sistema di tipi. Per quanto riguarda l'Esempio 20, la situazione è semplice: dovremmo avere

$\Gamma(a) = [T_1, T_2]$ e $\Gamma(a) = [T]$, e ciò è impossibile. Per quanto riguarda l'Esempio 21, abbiamo bisogno del nuovo sistema di tipi per rifiutare il processo $a(x).x(z) \mid \overline{a}\langle c\rangle.\overline{c}\langle b, b\rangle$. Per (T-Par), (T-In), (T-Out) e (T-Nil), abbiamo

$$\dfrac{\Gamma(a) = [T] \qquad \dfrac{(\Gamma, x : T)(x) = [T_1] \qquad \Gamma, x : T, z : T_1 \vdash \mathbf{0}}{\Gamma, x : T \vdash x(z).\mathbf{0}}}{\Gamma \vdash a(x).x(z).\mathbf{0}}$$

e

$$\dfrac{\Gamma(a) = [T] \qquad \Gamma(c) = T \qquad \dfrac{\Gamma(c) = [T', T'] \qquad \Gamma(b) = T' \qquad \Gamma \vdash \mathbf{0}}{\Gamma \vdash \overline{c}\langle b, b\rangle.\mathbf{0}}}{\overline{a}\langle c\rangle.\overline{c}\langle b, b\rangle.\mathbf{0}}$$

La prima derivazione implica $T = [T_1]$, mentre la seconda derivazione implica $T = [T', T']$. Pertanto, la derivazione mostrata è inconsistente poichè $[T', T'] \neq [T_1]$.

Passiamo ora a dimostrare la correttezza del sistema di tipi. A tal scopo dobbiamo dimostrare due risultati: *subject reduction*, che assicura che la tipabilità è un'invariante delle riduzioni, e *type safety*, che assicura che termini tipabili non generano errori runtime. Per far ciò, abbiamo bisogno di alcuni lemmi tecnici che andiamo ora ad enunciare e provare.

Lemma 22 (Substitution) *Siano* $\Gamma \vdash P$, $\Gamma = \Gamma', x : T$ *e* $\Gamma'(b) = T$*; allora* $\Gamma' \vdash P[b/x]$.

Dim.: per induzione sulla profondità dell'inferenza di $\Gamma \vdash P$.

Passo base: l'unica regola senza premesse è (T-Nil); pertanto $P \triangleq \mathbf{0}$ e $\mathbf{0}[b/x] \triangleq \mathbf{0}$ da cui, per (T-Nil), $\Gamma' \vdash \mathbf{0}$.

Passo induttivo: distinguiamo l'ultima regola usata.

(T-In) Per definizione della regola, abbiamo che $P \triangleq a(x_1, \ldots, x_k).P'$, $\Gamma(a) \triangleq [T_1, \ldots, T_k]$, $\Gamma, x_1 : T_1, \ldots, x_k : T_k \vdash P'$ e $\{x_1, \ldots, x_k\} \cap dom(\Gamma) = \emptyset$; da quest'ultimo fatto segue che $x \notin \{x_1, \ldots, x_k\}$ e che $\{x_1, \ldots, x_k\} \cap dom(\Gamma') = \emptyset$. Per induzione, $\Gamma', x_1 : T_1, \ldots, x_k : T_k \vdash P'[b/x]$. Se $x \neq a$ allora $\Gamma(a) = \Gamma'(a) = [T_1, \ldots, T_k]$ e quindi, per (T-In), $\Gamma' \vdash a(x_1, \ldots, x_k).(P'[b/x])$; ciò implica $\Gamma' \vdash (a(x_1, \ldots, x_k).P')[b/x]$. Se $x = a$ allora $\Gamma(x) = \Gamma(a) = \Gamma'(b) = [T_1, \ldots, T_k]$ e quindi, per (T-In), $\Gamma' \vdash b(x_1, \ldots, x_k).(P'[b/x])$; ciò implica $\Gamma' \vdash (a(x_1, \ldots, x_k).P')[b/x]$.

(T-OUT): simile.

(T-RES): in questo caso, $P \triangleq (\nu a)P'$, $\Gamma, a : T' \vdash P'$ e $a \notin dom(\Gamma)$; quest'ultimo fatto implica che $x \neq a$ e che $a \notin dom(\Gamma')$. Per induzione, $\Gamma', a : T' \vdash P'[b/x]$ e, per (T-RES), $\Gamma' \vdash (\nu a)(P'[b/x])$; ciò implica $\Gamma' \vdash ((\nu a)P')[b/x]$.

(T-PAR), (T-MATCH), (T-REP): semplice induzione. ∎

Le dimostrazioni dei Lemmi 23 e 24 sono per induzione sulla profondità dell'inferenza di $\Gamma \vdash P$ e vengono lasciate come facile esercizio al lettore.

Lemma 23 (Weakening) *Se* $\Gamma \vdash P$ *e* $a \notin dom(\Gamma) \cup n(P)$, *allora* $\Gamma, a : T \vdash P$.

Lemma 24 (Strenghtening) *Se* $\Gamma, a : T \vdash P$ *e* $a \notin fn(P)$ *allora* $\Gamma \vdash P$.

Lemma 25 (Subject Congruence) *Se* $\Gamma \vdash P$ *e* $P \equiv Q$ *allora* $\Gamma \vdash Q$.

Dim.: per mutua induzione sulla profondità dell'inferenza di $P \equiv Q$ e $Q \equiv P$. Molti casi nel passo base sono banali:

- (S-REFL) e (S-RCOM) sono immediati;
- (S-ID), (S-ASS), (S-COM) si basano sulle regole (T-NIL) e (T-PAR);
- (S-REP) si basa su (T-REP) e (T-PAR);
- (S-EQ) si basa su (T-MATCH);
- (S-ABS) si basa su (T-NIL);
- (S-CONV) dipende dal fatto che ridenominare nomi bound non crea problemi, grazie al Lemma 24;
- L'unico caso non banale si ha quando si usa (S-EXT). In tal caso, $P \triangleq P_1 \mid (\nu n)P_2$, $Q \triangleq (\nu n)(P_1 \mid P_2)$ e $n \notin fn(P_1)$. Per come è stato definito il type system, applicando (T-RES) e (T-PAR), abbiamo che:

$$\frac{\Gamma \vdash P_1 \qquad \dfrac{\exists T.\Gamma, n : T \vdash P_2}{\Gamma \vdash (\nu n)P_2}\ n \notin dom(\Gamma)}{\Gamma \vdash P_1 \mid (\nu n)P_2}$$

Per il Lemma 23 si ha che $\Gamma, n : T \vdash P_1$; quindi applicando (T-PAR) e (T-RES)

$$\frac{\dfrac{\Gamma, n : T \vdash P_1 \qquad \Gamma, n : T \vdash P_2}{\Gamma, n : T \vdash P_1 \mid P_2}}{\Gamma \vdash (\nu n)(P_1 \mid P_2)}$$

I quattro casi induttivi (S-Trans), (S-Simm), (S-Res) e (S-Par) si dimostrano con una semplice induzione. ∎

Teorema 26 (Subject Reduction) *Se* $\Gamma \vdash P$ *e* $P \longmapsto P'$ *allora* $\Gamma \vdash P'$.

Dim.: per induzione sulla profondità dell'inferenza $P \longmapsto P'$.

Passo base: l'unica regola senza premesse è (R-Com). In tal caso, $P \triangleq a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{b}\rangle.P_2$ e $P' \triangleq P_1[\widetilde{b}/\widetilde{x}] \mid P_2$. Per come è stato definito il type system abbiamo che, utilizzando (T-Par):

$$\frac{\Gamma \vdash a(\widetilde{x}).P_1 \qquad \Gamma \vdash \overline{a}\langle\widetilde{b}\rangle.P_2}{\Gamma \vdash P}$$

Utilizzando (T-In), si ottiene

$$\frac{\Gamma(a) = [T_1, \ldots, T_k] \qquad \Gamma, x_1 : T_1, \ldots, x_k : T_k \vdash P_1}{\Gamma \vdash a(x_1, \ldots, x_k).P_1}$$

per $\widetilde{x} = x_1, \ldots, x_k$ e $\{x_1, \ldots, x_k\} \cap dom(\Gamma) = \emptyset$; utilizzando (T-Out), si ottiene

$$\frac{\Gamma(a) = [T_1, \ldots, T_k] \qquad \forall i = 1, \ldots, h.\Gamma(b_i) = T_i \qquad \Gamma \vdash P_2}{\Gamma \vdash \overline{a}\langle b_1, \ldots, b_h\rangle.P_2}$$

per $\widetilde{b} = b_1, \ldots, b_k$. Applicando il Lemma 22 k volte, abbiamo che $\Gamma, x_1 : T_1, \ldots, x_k : T_k \vdash P_1$ e $\forall i = 1, \ldots, k.\Gamma(b_i) = T_i$ implicano $\Gamma \vdash P_1[\widetilde{b}/\widetilde{x}]$. Pertanto, per (T-Par), è facile concludere $\Gamma \vdash P'$.

Passo induttivo: distinguiamo l'ultima regola usata.

(R-Par) In questo caso, $P \triangleq P_1 \mid P_2$, $P_1 \longmapsto P_1'$ e $P' \triangleq P_1' \mid P_2$. Inoltre, $\Gamma \vdash P$ implica $\Gamma \vdash P_1$ e $\Gamma \vdash P_2$. Allora, per induzione, $\Gamma \vdash P_1'$; da ciò concludiamo, per (T-Par), che $\Gamma \vdash P'$.

(R-Res) Simile.

(R-Struct) In questo caso, $P \equiv Q$, $Q \longmapsto Q'$ e $Q' \equiv P'$. Per il Lemma 25, $\Gamma \vdash Q$ che, per induzione, implica $\Gamma \vdash Q'$. Ancora per il Lemma 25, si ottiene $\Gamma \vdash P'$. ∎

Teorema 27 (Type Safety) *Se* $\exists\Gamma.\Gamma \vdash P$ *allora* $P \not\downarrow$.

Dim.: dimostriamo il contronominale, cioè che se $P \uparrow$ allora $\nexists \Gamma.\Gamma \vdash P$. Questo è fatto per induzione sulla lunghezza dell'inferenza di $P \uparrow$.

Passo base: l'unica regola applicabile è (E-Com). In tal caso, $P \triangleq a(\widetilde{x}).P_1 \mid \overline{a}\langle\widetilde{b}\rangle.P_2$ per $k = |\widetilde{x}| \neq |\widetilde{b}| = h$. Assumiamo per assurdo che $\exists \Gamma.\Gamma \vdash P$. Per (T-Par) abbiamo che $\Gamma \vdash a(\widetilde{x}).P_1$ e $\Gamma \vdash \overline{a}\langle\widetilde{b}\rangle.P_2$; ciò implicherebbe $\Gamma(a) = [T_1, \ldots, T_k]$ e $\Gamma(a) = [T'_1, \ldots, T'_h]$. Ma questo è un assurdo poichè $k \neq h$ e poichè Γ è una funzione.

Passo induttivo: distinguo l'ultima regola usata.

(E-Par)**:** In questo caso, $P \triangleq P_1 \mid P_2$ con $P_1 \uparrow$. Per induzione $\nexists \Gamma.\Gamma \vdash P_1$ allora non è possibile applicare (T-Par) e quindi $\nexists \Gamma.\Gamma \vdash P_1 \mid P_2$.

(E-Res)**:** simile.

(E-Struct)**:** in questo caso, $P \equiv Q$ con $Q \uparrow$. Supponiamo per assurdo che $\Gamma \vdash P$. Allora, essendo $P \equiv Q$, per il Lemma 25 avremmo $\Gamma \vdash Q$, in contraddizione con $\nexists \Gamma.\Gamma \vdash Q$, che segue per induzione. ∎

In conclusione, possiamo ottenere il risultato desiderato, cioè che processi tipabili non generano mai errori a run-time.

Corollario 28 *Se P è tipabile (cioè, se esiste un Γ tale che $\Gamma \vdash P$), allora $P' \not\uparrow$, per ogni P' tale che $P \longmapsto^* P'$.*

Dim.: Sia $P \longmapsto^n P'$; la prova è per induzione su n. Il caso base ($n = 0$ e quindi $P' \triangleq P$) è il Teorema 27; il caso induttivo segue da una semplice applicazione dei Teoremi 26 e 27. ∎

2.1.3 Il π-calcolo Poliadico nel π-calcolo Monadico

Confrontando l'Esempio 7 con l'Esempio 19, abbiamo visto che il π-calcolo poliadico è un formalismo che permette di scrivere processi in modo più compatto. Ora il problema che ci si pone è quello di capire se il π-calcolo poliadico è solo notazionalmente più comodo da usare o se ha un maggiore potere espressivo. In altre parole è possibile esprimere il π-calcolo poliadico tramite il π-calcolo monadico?

Per rispondere a questa domanda, ci concentriamo su una forma semplificata del π-calcolo poliadico: il π-calcolo *biadico*. Nel π-calcolo biadico, ogni

comunicazione coinvolge sempre e solo 2 nomi; pertanto la regola di fondo su cui si basa il π-calcolo biadico è:

$$a(x,y).P \;\mid\; \overline{a}\langle b,c\rangle.Q \longmapsto P[{}^{b,c}\!/_{x,y}] \;\mid\; Q$$

Ora passiamo a definire una codifica da π-calcolo biadico a π-calcolo monadico. Essa è una funzione $(\!|\cdot|\!) : \pi_{bi} \longrightarrow \pi_{mon}$ definita induttivamente sulla sintassi dei processi come segue:

1. $(\!|\mathbf{0}|\!) \triangleq \mathbf{0}$
2. $(\!|(\nu a)P|\!) \triangleq (\nu a)(\!|P|\!)$
3. $(\!|P \mid Q|\!) \triangleq (\!|P|\!) \;\mid\; (\!|Q|\!)$
4. $(\!|[a=b]P|\!) \triangleq [a=b](\!|P|\!)$
5. $(\!|!P|\!) \triangleq !\,(\!|P|\!)$
6. $(\!|\overline{a}\langle b,c\rangle.P|\!) \triangleq (\nu d)\overline{a}\langle d\rangle.\overline{d}\langle b\rangle.\overline{d}\langle c\rangle.(\!|P|\!)$, per $d \notin fn(a,b,c,P)$
7. $(\!|a(x,y).P|\!) \triangleq a(z).z(x).z(y).(\!|P|\!)$, per $z \notin fn(a,x,y,P)$

I primi cinque casi sono un semplice omomorfismo. La codifica di un output (sesto caso), invece, prende un nome nuovo, lo dichiara ristretto e lo trasmette su a; su tale canale ristretto vengono poi passati in sequenza b e c. Si noti che la restrizione del canale garantisce l'assenza di interferenze nella trasmissione di b e c. Simmetricamente, nel codificare un input (settimo caso), si riceve un canale da cui giungeranno i nomi, uno alla volta.

Passiamo ora a studiare le proprietà di cui gode tale encoding e a ragionare in maniera formale sul potere espressivo dei due calcoli. Una misura ragionevole di ciò potrebbe essere il costo computazionale di un azione nel processo poliadico e nel corrispondente processo monadico. In tal caso, vediamo che un output/input biadico è mimato da tre output/input monadici. Generalizzando possiamo dire che una singola azione poliadica n-argomentale è mimata da $n+1$ azioni monadiche; questo è molto vicino all'encoding di efficienza massima, visto che ogni comunicazione n-argomentale deve essere mimata da almeno n comunicazioni monadiche.

Un altro possibile approccio consiste nel ragionare sul comportamento dei processi. Infatti vogliamo che la funzione di codifica sia "fedele", cioè mantenga il comportamento dei processi originari. Qui presentiamo una proprietà 'minimale' chiamata *corrispondenza operazionale*. Indicando con $\longmapsto^*$ zero o più passi di riduzione, essa può essere formulata come segue.

Proposizione 29 *Se* $P \longmapsto P'$*, allora* $(\!|P|\!) \longmapsto^* (\!|P'|\!)$.

Proposizione 30 *Se* $(\!|P|\!) \longmapsto Q$*, allora esiste* P' *tale che* $P \longmapsto^* P'$ *e* $Q \longmapsto^* (\!|P'|\!)$.

La prima proprietà assicura che se un processo poliadico P effettua una comunicazione ed evolve in P', il corrispondente processo monadico effettuerà una o più comunicazioni e si ridurrà all'encoding di P'. In altre parole il comportamento del processo sorgente viene mimato dal processo destinazione. Però solo questo non basta; infatti vogliamo anche essere 'conservativi', cioè vogliamo che ogni comportamento del processo tradotto sia in qualche modo originato dal processo sorgente. Questo viene formalizzato dalla seconda proprietà. Si noti che Q non è in generale l'encoding di un processo poliadico. L'unica cosa che possiamo richiedere è che esso possa ridursi all'encoding di un opportuno termine poliadico.

La proprietà della corrispondenza operazionale, pur non essendo una proprietà forte, è basilare ed indispensabile: cioè, è la minima proprietà che una codifica deve possedere. Va però fatto osservare che, da sola, è poco informativa: anche l'encoding che traduce tutto in $\mathbf{0}$ gode di tale proprietà. Per tale encoding valgono proprietà più forti; per una discussione più dettagliata, si veda [6].

Dim.: (di Proposizione 29) per induzione sulla lunghezza dell'inferenza di $P \longmapsto P'$.

Passo Base: l'unica regola utilizzabile è (R-Com). In tal caso, $P \triangleq a(x,y).P_1 \mid \overline{a}\langle b,c\rangle.P_2$ e $P' \triangleq P_1[b,c/x,y] \mid P_2$. Per definizione di encoding abbiamo che $(\!|P|\!) \triangleq a(z).z(x).z(y).(\!|P_1|\!) \mid (\nu d)\overline{a}\langle d\rangle.\overline{d}\langle b\rangle.\overline{d}\langle c\rangle.(\!|P_2|\!)$, per d e z fresh. Ora,

$$\begin{array}{rcl}(\!|P|\!) & \longmapsto & (\nu d)(d(x).d(y).(\!|P_1|\!) \mid \overline{d}\langle b\rangle.\overline{d}\langle c\rangle.(\!|P_2|\!)) \\ & \longmapsto & (\nu d)(d(y).(\!|P_1|\!)[b/x] \mid \overline{d}\langle c\rangle.(\!|P_2|\!)) \\ & \longmapsto & (\nu d)((\!|P_1|\!)[b,c/x,y] \mid (\!|P_2|\!)) \\ & \equiv & (\!|P_1|\!)[b,c/x,y] \mid (\!|P_2|\!) \quad \triangleq \quad (\!|P_1[b,c/x,y]|\!) \mid (\!|P_2|\!) \quad \triangleq \quad (\!|P'|\!)\end{array}$$

Infatti è facile dimostrare, per induzione strutturale su P, che la funzione di sostituzione e la funzione di encoding commutano.

Passo Induttivo: distinguiamo l'ultima regola usata.

(R-Par): in tal caso, $P \triangleq P_1 \mid Q$, $P_1 \longmapsto P_2$ e $P' \triangleq P_2 \mid Q$. Per induzione $(\!|P_1|\!) \longmapsto^* (\!|P_2|\!)$. Con una semplice dimostrazione per induzione sul numero di passi di riduzione in $\longmapsto^*$ è semplice mostrare che $(\!|P|\!) \triangleq (\!|P_1|\!) \mid (\!|Q|\!) \longmapsto^* (\!|P_2|\!) \mid (\!|Q|\!) \triangleq (\!|P'|\!)$.

(R-Res): simile al caso precedente.

(R-Struct): in tal caso, $P \equiv Q$, $Q \longmapsto Q'$ e $Q' \equiv P'$. Visto che l'equivalenza strutturale lavora su operazioni che l'encoding lascia inalterate, possiamo osservare che $\langle\!|P|\!\rangle \equiv \langle\!|Q|\!\rangle$ e $\langle\!|Q'|\!\rangle \equiv \langle\!|P'|\!\rangle$. Inoltre, per induzione abbiamo che $\langle\!|Q|\!\rangle \longmapsto^* \langle\!|Q'|\!\rangle$. Da ciò segue facilmente che $\langle\!|P|\!\rangle \longmapsto^* \langle\!|P'|\!\rangle$. ∎

Dim.: (di Proposizione 30) per induzione sulla lunghezza dell'inferenza di $\langle\!|P|\!\rangle \longmapsto Q$.

Passo Base: poichè $\langle\!|P|\!\rangle$ è un encoding, non potrà avere la forma $a(z).P_1 \mid \overline{a}\langle d\rangle.P_2$, come sarebbe necessario per usare (R-Com); al più sarà $\langle\!|P|\!\rangle \triangleq a(z).P_1 \mid (\nu d)\overline{a}\langle d\rangle.P_2$. Quindi la minima inferenza per $\langle\!|P|\!\rangle \longmapsto Q$ è

$$\dfrac{a(z).P_1 \mid (\nu d)\overline{a}\langle d\rangle.P_2 \;\equiv\; \dfrac{\dfrac{}{a(z).P_1 \mid \overline{a}\langle d\rangle.P_2 \longmapsto P_1[d/z] \mid P_2}}{(\nu d)(a(z).P_1 \mid \overline{a}\langle d\rangle.P_2) \longmapsto (\nu d)(P_1[d/z] \mid P_2)}}{a(z).P_1 \mid (\nu d)\overline{a}\langle d\rangle.P_2 \longmapsto (\nu d)(P_1[d/z] \mid P_2)}$$

ottenuta con una applicazione della (R-Struct) (per estendere lo scopo del nome d che, essendo fresh, non occorre in $a(z).P_1$) e una applicazione della (R-Res) e della (R-Com)(per effettuare la comunicazione all'interno dello scopo della restrizione). A questo punto, sempre poichè stiamo lavorando con $\langle\!|P|\!\rangle$ abbiamo che $P_1 \triangleq z(x).z(y).\langle\!|P_1'|\!\rangle$ e $P_2 \triangleq \overline{d}\langle b\rangle.\overline{d}\langle c\rangle.\langle\!|P_2'|\!\rangle$. Ma allora $P \triangleq a(x,y).P_1' \mid \overline{a}\langle b,c\rangle.P_2'$; pertanto, $P \longmapsto P_1'[b,c/x,y] \mid P_2' \triangleq P'$ e

$$\begin{aligned} Q \;\triangleq\; (\nu d)(d(x).d(y).\langle\!|P_1'|\!\rangle \mid \overline{d}\langle b\rangle.\overline{d}\langle c\rangle.\langle\!|P_2'|\!\rangle) \\ \longmapsto\longmapsto \equiv\; \langle\!|P_1'[b,c/x,y] \mid P_2'|\!\rangle \;\triangleq\; \langle\!|P'|\!\rangle \end{aligned}$$

Passo Induttivo: simile al passo induttivo della dimostrazione della Proposizione 29. ∎

Abbiamo dimostrato che il π-calcolo biadico si può codificare nel π-calcolo monadico e che questa codifica rispetta le proprietà minime che una codifica deve avere (le computazioni di un processo poliadico e del suo encoding sono in corrispondenza uno a uno tra loro). Va però fatto notare che l'encoding funziona bene se il processo poliadico di partenza è ben-tipato (nel caso del calcolo biadico questo è ovviamente vero); infatti, in [6] si mostra che termini non tipabili del π-calcolo poliadico non ammettono un encoding che rispetti un insieme di proprietà 'ragionevoli'.

Inoltre, [21] mostra che l'encoding $\langle\!|\cdot|\!\rangle$ non gode di proprietà semantiche più forti, tipo la *full abstraction*: $P \sim Q$ se e solo se $[\![P]\!] \sim [\![Q]\!]$. Questa proprietà rafforza la corrispondenza operazionale nel senso che tutti i comportamenti, non solo quelli attuali ma anche quelli potenziali, sono rappresentati e riflessi nell'encoding. Per la codifica del poliadico nel monadico questa proprietà *non* vale. Infatti, mentre è vero che $[\![P]\!] \sim [\![Q]\!]$ implica $P \sim Q$, il viceversa non vale. Consideriamo

$$P \triangleq \overline{a}\langle b, b\rangle.\overline{a}\langle b, b\rangle \qquad\qquad Q \triangleq \overline{a}\langle b, b\rangle \mid \overline{a}\langle b, b\rangle$$

I processi sono bisimili ma i loro encoding no. Si supponga per assurdo che lo siano. Allora, poichè $\sim$ è una congruenza rispetto al parallelo (vedi Proposizione 14), dovrebbe essere

$$[\![P]\!] \mid a(x) \sim [\![Q]\!] \mid a(x)$$

Ma questo non è vero. Infatti, consideriamo la sfida

$$[\![Q]\!] \mid a(x) \xrightarrow{\tau} (\nu c)\overline{c}\langle b\rangle.\overline{c}\langle b\rangle \mid [\![\overline{a}\langle b\rangle]\!] \sim [\![\overline{a}\langle b\rangle]\!]$$

L'unica possibile risposta è

$$[\![P]\!] \mid a(x) \xrightarrow{\tau} (\nu c)\overline{c}\langle b\rangle.\overline{c}\langle b\rangle.[\![\overline{a}\langle b\rangle]\!] \sim \mathbf{0}$$

in cui l'input da un canale ristretto rende il processo equivalente a $\mathbf{0}$. Ora, poichè $[\![\overline{a}\langle b\rangle]\!] \not\sim \mathbf{0}$, abbiamo che $[\![P]\!] \not\sim [\![Q]\!]$.

2.2 Il π-calcolo Asincrono

Come già detto in precedenza, la regola (R-Com) è il cuore della semantica del π-calcolo. Essa è caratterizzata da:

input bloccante: il processo P prefissato da un input è bloccato finchè questo non viene eseguito. P è effettivamente sospeso in quanto il suo comportamento dipende dal valore ricevuto in input.

interazione tra processi: la comunicazione è il risultato di attività complementari tra due processi paralleli: un processo invia un messaggio e l'altro lo riceve. Siamo di fronte ad una cooperazione tra due processi che avviene sfruttando un canale come mezzo di comunicazione.

output bloccante: il processo Q è sospeso finchè non riesce a spedire il suo messaggio.

Mentre ha senso che P attenda che la comunicazione sia terminata prima di procedere, l'output bloccante è meno giustificabile: il comportamento di chi spedisce non sempre dipende dallo scambio di informazione ma solo da un esito positivo della comunicazione. Esempi in tal senso possono essere il meccanismo di spedizione di un'e-mail, le informazioni messe su una pagina web o le trasmissioni radio e TV. Sono questi esempi tipici di comunicazione *asincrona* in cui, cioè, mittente e destinatario non si sincronizzano per scambiarsi messaggi. In pratica, l'atto di spedire e di ricevere un messaggio avvengono in due momenti diversi.

Nel π-calcolo si può modellare l'asincronia in due modi diversi. Si può sostituire la (R-Com) con le due regole

$$\text{(R-Snd)} \quad \overline{a}\langle b\rangle.P \longmapsto \overline{a}\langle b\rangle \mid P \qquad\qquad \text{(R-Rcv)} \quad a(x).P \mid \overline{a}\langle b\rangle \longmapsto P[{}^{b}\!/\!_{x}]$$

Questa soluzione, benchè realistica, non viene effettivamente utilizzata: c'è un passo per spedire ed un passo per ricevere. Il problema è che nel π-calcolo le computazioni sono comunicazioni e dividere questa comunicazione in due azioni separate snatura in parte l'intuizione alla base del linguaggio.

Si può allora provare a modificare leggermente la sintassi del π-calcolo in modo da proibire prefissi di output:

$$P ::= \mathbf{0} \mid a(x).P \mid \overline{a}\langle b\rangle \mid P_1|P_2 \mid (\nu n)P \mid !P \mid [a=b]P$$

In questo modo è sintatticamente impossibile mettere un processo in attesa dell'esecuzione di un'azione di output. A questo punto la regola (R-Com) è riformulata come:

$$a(x).P \mid \overline{a}\langle b\rangle \longmapsto P[{}^{b}\!/\!_{x}]$$

Questa soluzione, introdotta in [8, 4], è preferita nella pratica poichè mantiene il principio secondo cui un unico assioma modella l'essenza della computazione.

Codifiche Ci domandiamo ora se il π-calcolo sincrono ed asincrono siano interscambiabili, cioè se uno può codificare l'altro e viceversa. Banalmente, il π-calcolo sincrono può codificare quello asincrono: basta imporre che tutti i processi prefissati da un output siano $\mathbf{0}$. Cerchiamo ora di capire se vale il viceversa, cioè se sia possibile codificare il π-calcolo sincrono con il π-calcolo asincrono. Citando da [18]: "I massaggi nei sistemi distribuiti possono essere passati in maniera sincrona o asincrona (...) In molti casi il passaggio sincrono di messaggi si può considerare come un caso particolare del passaggio asincrono". Pertanto è sperabile che un tale encoding esista. Una possibilità,

proposta in [4], è la seguente:

$$\begin{array}{rcll}
[\![\mathbf{0}]\!] & \triangleq & \mathbf{0} & \\
[\![P_1|P_2]\!] & \triangleq & [\![P_1]\!] \mid [\![P_2]\!] & \\
[\![!P]\!] & \triangleq & ![\![P]\!] & \\
[\![(\nu n)P]\!] & \triangleq & (\nu n)[\![P]\!] & \\
[\![[a=b]P]\!] & \triangleq & [a=b][\![P]\!] & \\
[\![\overline{a}\langle b\rangle.P]\!] & \triangleq & (\nu c)(\overline{a}\langle c\rangle \mid c(y).(\overline{y}\langle b\rangle \mid [\![P]\!])) & \text{per } c, y \notin fn(P) \\
[\![a(x).P]\!] & \triangleq & a(z).(\nu d)(\overline{z}\langle d\rangle \mid d(x).[\![P]\!]) & \text{per } d, z \notin fn(P)
\end{array}$$

Per spiegare intuitivamente la definizione di tale encoding, ragioniamo per approssimazioni successive. La codifica più banale di $a(x).P \mid \overline{a}\langle b\rangle.Q$ è

$$a(x).P \mid \overline{a}\langle b\rangle \mid Q$$

Ma questa scelta non è soddisfacente perchè si dovrebbe bloccare Q finchè non c'è stata una comunicazione. Proviamo allora a prefissare Q con un input (bloccante):

$$a(x).(\overline{c}\langle d\rangle \mid P) \;\mid\; \overline{a}\langle b\rangle \;\mid\; c(y).Q$$

per $y \notin fn(Q)$. Questa soluzione è migliore, ma non ancora soddisfacente: infatti, se si mette questo termine in un contesto che conosce c, Q potrebbe rimanere bloccato per sempre (assumendo che nè P nè R spediscono più dati su c):

$$\begin{array}{c}
a(x).(\overline{c}\langle d\rangle \mid P) \;\mid\; \overline{a}\langle b\rangle \;\mid\; c(y).Q \;\mid\; c(z).R \\
\longmapsto \overline{c}\langle d\rangle \mid P[{}^{b}\!/\!_{x}] \mid c(y).Q \mid c(z).R \\
\longmapsto P[{}^{b}\!/\!_{x}] \mid c(y).Q \mid R[{}^{d}\!/\!_{z}]
\end{array}$$

oppure Q potrebbe essere sbloccato prima che il messaggio sia consumato:

$$a(x).(\overline{c}\langle d\rangle \mid P) \mid \overline{a}\langle b\rangle \mid c(y).Q \mid \overline{c}\langle d\rangle.R \longmapsto a(x).(\overline{c}\langle d\rangle \mid P) \mid \overline{a}\langle b\rangle \mid Q[{}^{d}\!/\!_{y}] \mid R$$

Dobbiamo quindi garantire che il canale su cui si manda l'ack sia noto solo a P e Q. Proviamo allora a tradurre $\overline{a}\langle b\rangle.Q$ in $(\nu c)(\overline{a}\langle c\rangle \mid c(x).Q \mid \overline{c}\langle b\rangle)$; ciò garantisce che l'ack non sia intercettabile dal contesto. Però così Q potrebbe considerare la spedizione di b come un ack e sbloccarsi.

Arriviamo così alla soluzione proposta da $[\![\cdot]\!]$, per cui $\overline{a}\langle b\rangle$ diventa "spedisci su a un nome fresh, resta in attesa di ricevere da tale nome e sul canale ricevuto spedisci b", mentre $a(x)$ diventa "ricevi da a, sul canale ricevuto spedisci un nome fresh (ack) e da tale nome leggi il vero messaggio".

Proprietà dell'encoding (Esercizio) Dimostrare che la corrispondenza operazionale (formulata come nelle Proposizioni 29 e 30) vale. Vale la proprietà di full abstraction? Se si dimostrarla, se no fornire un contresempio. L'encoding studiato simula una singola comunicazione sincrona con tre comunicazioni asincrone; è possibile dare un encoding che simuli una comunicazione sincrona con due comunicazioni asincrone?

2.3 Il π-calcolo Higher-order

Il π-calcolo modella meglio del CCS vari aspetti di sistemi distribuiti; però ancora ci sono aspetti che non si modellano in maniera naturale. Per esempio, il contenuto di un messaggio può essere codice eseguibile: e-mail con attachment, applet java scaricata da una pagina web, remote code evaluation (spedisco un programma per l'esecuzione su un server che poi restituisce un risultato). Per questo introduciamo il π-calcolo *higher-order*.

Sia X una generica variabile di processo scelta da un insieme $\mathcal{V}$ disgiunto da $\mathcal{N}$. La sintassi del π-calcolo higher-order è:

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid P_1|P_2 \mid (\nu n)P \mid [a=b]P \mid a(\theta).P \mid \overline{a}\langle\eta\rangle.P \mid X \\ \eta & ::= & b \mid P \\ \theta & ::= & x \mid X \end{array}$$

La semantica a riduzioni è estesa in maniera banale ponendo (R-Com)

$$a(\theta).P \mid \overline{a}\langle\eta\rangle.Q \longmapsto P[{}^{\eta}/_{\theta}] \mid Q$$

Ad esempio $a(X).(X \mid X) \mid \overline{a}\langle\overline{b}\langle c\rangle\rangle \longmapsto \overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle$. E' chiaramente necessario un sistema di tipi per evitare situazioni tipo $a(X).(X \mid X) \mid \overline{a}\langle b\rangle$ o $a(x).x(y) \mid \overline{a}\langle\overline{b}\langle c\rangle\rangle$.

Esercizio: sviluppare un sistema di tipi che eviti anomalie di questo genere, se ne definisca formalmente l'errore runtime e si dimostri che programmi ben tipati non generano mai errori runtime.

Notiamo che della sintassi manca l'operatore di replicazione. Questo non è un problema: grazie al passaggio di processi come argomenti in una comunicazione, la replicazione è implementabile come segue:

$$\begin{array}{rcll} D & \triangleq & a(X).(X \mid \overline{a}\langle X\rangle) & & \text{Duplicatore} \\ R(P) & \triangleq & (\nu a)(D \mid \overline{a}\langle P \mid D\rangle) & \text{per } a \text{ fresh} & \text{Replicatore} \\ & \longmapsto & (\nu a)(P \mid D \mid \overline{a}\langle P \mid D\rangle) & & \\ & \equiv & P \mid (\nu a)(D \mid \overline{a}\langle P \mid D\rangle) & \triangleq \quad P \mid R(P) & \end{array}$$

L'unica differenza di rilievo rispetto alla regola (S-Rep) è che ora l'attivazione di ogni copia di P richiede un passo di riduzione mentre, usando (S-Rep) e (R-Struct), si possono attivare in una sola riduzione più copie di P simultaneamente.

Abbastanza sorprendentemente il π-calcolo higher-order si può codificare nel π-calcolo first-order. Dato un processo P del π-calcolo higher-order, prendiamo una funzione parziale $\varphi : \mathcal{V} \to \mathcal{N}$ tale che $imm(\varphi) \cap n(P) = \emptyset$. Ora l'encoding è:

$$\begin{array}{rcll} [\![a(X).P]\!] & \triangleq & a(x).[\![P]\!] & \text{per } x = \varphi(X) \\ [\![\overline{a}\langle P\rangle.Q]\!] & \triangleq & (\nu n)\overline{a}\langle n\rangle.([\![Q]\!] \mid !n().[\![P]\!]) & \text{per } n \notin fn(P,Q) \cup imm(\varphi) \\ [\![X]\!] & \triangleq & \overline{x}\langle\rangle & \text{per } x = \varphi(X) \end{array}$$

dove con $n()$ e $\overline{x}\langle\rangle$ intendiamo che l'argomento dell'input e dell'output è irrilevante.

L'idea alla base dell'encoding è la seguente: invece di spedire P, spediamo un riferimento a P, cioè un nome nuovo univocamente associato a P (viz. n); possiamo vedere P come il corpo di una funzione il cui nome è il nome nuovo. Una volta spedito, P deve essere bloccato finchè qualcuno non lo invoca; seguendo l'idea che "bloccato" si ottiene con un prefisso di input, abbiamo che P deve essere prefissato con un input dal nome che è il riferimento a P. Una volta ricevuto, P può essere usato più volte (per esempio, duplicato o rispedito); pertanto, la traduzione di P deve essere replicata per modellare la disponibilità di un numero arbitrario di copie di P. Una attivazione di P è quindi un output sul canale che è il riferimento di P.

Mostriamo intuitivamente come tale encoding goda della corrispondenza operazionale. A tal fine, consideriamo come esempio il termine

$$a(X).(X \mid X) \mid \overline{a}\langle\overline{b}\langle c\rangle\rangle$$

Facendone l'encoding otteniamo:

$$\begin{array}{rl} & a(x).(\overline{x}\langle\rangle \mid \overline{x}\langle\rangle) \;\mid\; (\nu n)\overline{a}\langle n\rangle.(\mathbf{0} \mid !n().\overline{b}\langle c\rangle) \\ \longmapsto & (\nu n)(\overline{n}\langle\rangle \mid \overline{n}\langle\rangle \mid !n().\overline{b}\langle c\rangle) \\ \longmapsto & (\nu n)(\overline{n}\langle\rangle \mid !n().\overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle) \\ \longmapsto & (\nu n)(!n().\overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle) \\ \equiv & (\nu n)(!n().\overline{b}\langle c\rangle) \mid \overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle \;\sim\; \overline{b}\langle c\rangle \mid \overline{b}\langle c\rangle \end{array}$$

In generale, si dimostri la seguente proprietà di corrispondenza operazionale:

- Se $P \longmapsto P'$, allora $[\![P]\!] \longmapsto^* \sim [\![P']\!]$.
- Se $[\![P]\!] \longmapsto Q$, allora esiste P' tale che $P \longmapsto^* P'$ e $Q \longmapsto^* \sim [\![P']\!]$.

Capitolo 3

Confronto tra Diversi Paradigmi

Per concludere, confrontiamo ora il paradigma di programmazione concorrente, di cui il π-calcolo è uno dei più noti rappresentanti, con altri ben noti paradigmi di programmazione. Partiremo con un confronto con il paradigma funzionale e passaremo poi al confronto con il paradigma ad oggetti.

3.1 Encoding del λ-calcolo nel π-calcolo

Come è ben noto, il λ-calcolo [2] è l'essenza dei linguaggi funzionali e, assieme alle funzioni ricorsive parziali e alle Macchine di Turing, è alla base dello studio della computabilità. Pertanto, esso ha sia un interesse teorico che pratico. La sua sintassi è:

$$M ::= x \quad | \quad \lambda x.M \quad | \quad MN$$

dove x è presa da un insieme numerabile di variabili. Una sua possibile semantica è quella lazy caratterizzata dalle seguenti regole:

$$(\lambda x.M)N \mapsto_\beta M[^N/_x] \qquad \frac{M \mapsto_\beta M'}{MN \mapsto_\beta M'N}$$

Alcuni λ-termini tipici sono i seguenti:

$I \triangleq \lambda x.x$ (Identità)
da cui $IN \mapsto_\beta N$

$D \triangleq \lambda x.xx$ (Autoapplicazione)
da cui $DN \mapsto_\beta NN$

$S \triangleq \lambda x.\lambda y.yx$ (Swapper)
da cui $SMN \mapsto_\beta (\lambda y.yM)N \mapsto_\beta NM$

Ci sono vari motivi per cui è interessante codificare il λ-calcolo nel π-calcolo: essendo il λ-calcolo Turing completo, un tale encoding mostrerebbe formalmente che anche il π-calcolo è Turing completo; inoltre, tale codifica permetterebbe di studiare implementazioni concorrenti di linguaggi funzionali, nonchè di confrontare due paradigmi diversi (le funzioni trasformano un input in un output e sono quindi inerentemente sequenziali; i processi, invece, offrono un'interazione tra attività indipendenti e pertanto sono inerentemente paralleli). Il λ-calcolo *passa funzioni*; quindi è di tipo higher-order ed è alla base del paradigma funzionale. Il π-calcolo *passa riferimenti*; quindi è di tipo first-order ed assomiglia al paradigma ad oggetti.

Codifica. Partiamo da un encoding intuitivo (molto simile all'encoding del π-calcolo higher-order mostrato in Sezione 2.3):

$$\begin{array}{rcll} [\![\lambda x.M]\!] & \triangleq & l(x).[\![M]\!] & \\ [\![MN]\!] & \triangleq & [\![M]\!] \mid (\nu c)\bar{l}\langle c\rangle.!c().[\![N]\!] & \text{per } c \text{ fresh} \\ [\![x]\!] & \triangleq & \overline{x}\langle\rangle & \end{array}$$

dove l è un nome riservato. Questo encoding in generale non funziona in quanto, a partire dalla seguente codifica del termine Swapper

$$\begin{array}{rcl} [\![S]\!] & \triangleq & l(x).l(y).(\overline{y}\langle\rangle \mid (\nu c_1)\bar{l}\langle c_1\rangle.!c_1().\overline{x}\langle\rangle) \\ [\![SM]\!] & \triangleq & [\![S]\!] \mid (\nu c_2)\bar{l}\langle c_2\rangle.!c_2().[\![M]\!] \\ [\![SMN]\!] & \triangleq & [\![SM]\!] \mid (\nu c_3)\bar{l}\langle c_3\rangle.!c_3().[\![N]\!] \end{array}$$

si può generare una derivazione 'sbagliata' rispetto al termine Swapper del λ-calcolo:

$$\begin{array}{l} [\![SMN]\!] \equiv \\ (\nu c_2 c_3)(\underline{l(x)}.l(y).(\overline{y}\langle\rangle \mid (\nu c_1)\bar{l}\langle c_1\rangle.!c_1().\overline{x}\langle\rangle) \mid \bar{l}\langle c_2\rangle.!c_2().[\![M]\!] \mid \underline{\bar{l}\langle c_3\rangle}.!c_3().[\![N]\!]) \\ \longmapsto (\nu c_2 c_3)(\underline{l(y)}.(\overline{y}\langle\rangle \mid (\nu c_1)\bar{l}\langle c_1\rangle.!c_1().\overline{c_3}\langle\rangle) \mid \underline{\bar{l}\langle c_2\rangle}.!c_2().[\![M]\!] \mid !c_3().[\![N]\!]) \\ \longmapsto (\nu c_1 c_2 c_3)(\underline{\overline{c_2}\langle\rangle} \mid \bar{l}\langle c_1\rangle.!c_1().\overline{c_3}\langle\rangle \mid \underline{!c_2()}.[\![M]\!] \mid !c_3().[\![N]\!]) \\ \longmapsto (\nu c_1 c_2 c_3)(\bar{l}\langle c_1\rangle.!c_1().\overline{c_3}\langle\rangle \mid !c_2().[\![M]\!] \mid [\![M]\!] \mid !c_3().[\![N]\!]) \\ \equiv [\![M]\!] \mid (\nu c_1)(\nu c_3)(\bar{l}\langle c_1\rangle.!c_1().\overline{c_3}\langle\rangle \mid !c_3().[\![N]\!]) \mid (\nu c_2)!c_2().[\![M]\!] \\ \sim [\![M]\!] \mid (\nu c_1)(\nu c_3)(\bar{l}\langle c_1\rangle.!c_1().\overline{c_3}\langle\rangle \mid !c_3().[\![N]\!]) \\ \approx [\![M]\!] \mid (\nu c_1)\bar{l}\langle c_1\rangle.!c_1().[\![N]\!] \triangleq [\![MN]\!] \end{array}$$

dove abbiamo sottolineato le coppie di prefissi che si sincronizzano per generare una riduzione. In questo caso abbiamo ottenuto l'encoding del termine

MN, invece che del termine NM, come avremmo voluto. Ovviamente, poichè il canale l prende in input sia il primo che il secondo argomento e poichè gli output di c_2 e c_3 sono fatti in parallello, sarebbe stato possibile ottenere anche $[\![NM]\!]$. L'encoding proposto mantiene tutte le computazioni del termine originale ma ne aggiunge di nuove; quindi non è una buona funzione di codifica.

Il problema di questo encoding è che ogni applicazione è codificata da una comunicazione sullo *stesso* canale l. Per correggere questo problema utilizziamo un encoding parametrico, specificando il canale su cui avviene la comunicazione. Ora, ogni applicazione avviene su un canale apposito, usato una sola volta. Inoltre utilizziamo un π-calcolo biadico per specificare il nome del canale su cui avviene la comunicazione e l'argomento da passare.

$$\begin{array}{rcll} [\![\lambda x.M]\!]_u & \triangleq & u(x,v).[\![M]\!]_v & \\ [\![MN]\!]_u & \triangleq & (\nu d)([\![M]\!]_d \mid (\nu c)\overline{d}\langle c,u\rangle.!c(w).[\![N]\!]_w) & \text{per } c,d \text{ fresh} \\ [\![x]\!]_u & \triangleq & \overline{x}\langle u\rangle & \end{array}$$

Vediamo ora che questo nuovo encoding risolve il problema dell'esempio precedente. Infatti, la nuova codifica del termine Swapper è

$$\begin{array}{rcl} [\![S]\!]_{d_2} & \triangleq & d_2(x,v).v(y,v').(\nu d_1)(\overline{y}\langle d_1\rangle \mid (\nu c_1)\overline{d_1}\langle c_1,v'\rangle.!c_1(w).\overline{x}\langle w\rangle) \\ [\![SM]\!]_{d_3} & \triangleq & (\nu d_2)([\![S]\!]_{d_2} \mid (\nu c_2)\overline{d_2}\langle c_2,d_3\rangle.!c_2(w).[\![M]\!]_w) \\ [\![SMN]\!]_u & \triangleq & (\nu d_3)([\![SM]\!]_{d_3} \mid (\nu c_3)\overline{d_3}\langle c_3,u\rangle.!c_3(w).[\![N]\!]_w) \end{array}$$

che può *solo* evolvere come segue:

$$\begin{array}{l} [\![SMN]\!] \equiv \\ (\nu c_2c_3d_2d_3)(\underline{d_2(x,v)}.v(y,v').(\nu d_1)(\overline{y}\langle d_1\rangle \mid (\nu c_1)\overline{d_1}\langle c_1,v'\rangle.!c_1(w).\overline{x}\langle w\rangle) \\ \qquad\qquad \mid \underline{\overline{d_2}\langle c_2,d_3\rangle}.!c_2(w).[\![M]\!]_w \mid \overline{d_3}\langle c_3,u\rangle.!c_3(w).[\![N]\!]_w) \\ \longmapsto (\nu c_2c_3d_3)(\underline{d_3(y,v')}.(\nu d_1)(\overline{y}\langle d_1\rangle \mid (\nu c_1)\overline{d_1}\langle c_1,v'\rangle.!c_1(w).\overline{c_2}\langle w\rangle) \\ \qquad\qquad \mid !c_2(w).[\![M]\!]_w \mid \underline{\overline{d_3}\langle c_3,u\rangle}.!c_3(w).[\![N]\!]_w) \\ \longmapsto (\nu c_1c_2c_3d_1)(\underline{\overline{c_3}\langle d_1\rangle} \mid \overline{d_1}\langle c_1,u\rangle.!c_1(w).\overline{c_2}\langle w\rangle \mid !c_2(w).[\![M]\!]_w \mid \underline{!c_3(w)}.[\![N]\!]_w) \\ \longmapsto (\nu c_1c_2c_3d_1)(\overline{d_1}\langle c_1,u\rangle.!c_1(w).\overline{c_2}\langle w\rangle \mid !c_2(w).[\![M]\!]_w \mid !c_3(w).[\![N]\!]_w \mid [\![N]\!]_{d_1}) \\ \equiv (\nu d_1)([\![N]\!]_{d_1} \mid (\nu c_1,c_2)(\overline{d_1}\langle c_1,u\rangle.!c_1(w).\overline{c_2}\langle w\rangle \mid !c_2(w).[\![M]\!]_w)) \mid (\nu c_3)!c_3(w).[\![N]\!]_w \\ \sim (\nu d_1)([\![N]\!]_{d_1} \mid (\nu c_1,c_2)(\overline{d_1}\langle c_1,u\rangle.!c_1(w).\overline{c_2}\langle w\rangle \mid !c_2(w).[\![M]\!]_w)) \\ \approx (\nu d_1)([\![N]\!]_{d_1} \mid (\nu c_1)\overline{d_1}\langle c_1,u\rangle.!c_1(w).[\![M]\!]_w) \\ \triangleq [\![NM]\!]_u \end{array}$$

Nelle derivazioni sopra ottenute possiamo sostituire ogni occorrenza di $\longmapsto$ con $\approx$. Infatti si può dimostrare il seguente risultato di correttezza (vedi [11]).

Teorema 31 $[\![(\lambda x.M)N]\!]_u \approx [\![N[^M/_x]]\!]_u$

3.2 Un Modello per Linguaggi a Oggetti

Per prima cosa osserviamo quali sono le caratteristiche del paradigma di programmazione orientato agli oggetti:

1. gli oggetti sono entità indipendenti in esecuzione parallela che interagiscono con scambi di messaggi;
2. gli oggetti sono basati sull'idea di *incapsulamento* (dati privati inaccessibili dall'esterno);
3. gli oggetti forniscono all'esterno delle funzionalità ed i messaggi servono ad attivarle;
4. un oggetto a può spedire un messaggio ad un oggetto b se a ha un riferimento per b;
5. gli oggetti sono dinamici, vengono cioè creati e distrutti a run-time;
6. i messaggi contengono informazioni da passare al metodo dell'oggetto invocato e questi parametri possono essere riferimenti ad altri oggetti;
7. gli oggetti vengono visti come istanze di una classe.

Il π-calcolo è evidentemente un ottimo candidato per modellare linguaggi ad oggetti ed è facile trovare analogie per ogni punto della lista sopra elencata:

1. processi paralleli del π-calcolo;
2. tramite l'operatore ν si ottengono canali riservati;
3. l'input modella l'offerta di una funzionalità e l'output una richiesta di attivazione;
4. due processi possono comunicare se entrambi condividono lo stesso canale;
5. tramite l'operatore ν si può modellare la creazione di oggetti e tramite (S-Abs) la loro distruzione;
6. le informazioni che i processi si scambiano sono a loro volta altri canali di comunicazione;

7. modellabile tramite l'operatore di replicazione.

Consideriamo ora la sintassi del seguente frammento di un linguaggio ad oggetti e, tramite un paio di esempi, mostriamo come programmi scritti in tale linguaggio possano essere tradotti in maniera naturale nel π-calcolo:

TIPI:
T ::= `void` | `int` | `ref`(A)

ESPRESSIONI:
E ::= X | $op(E_1.....E_n)$ | `new`(A) | $E.m(E_1, \ldots, E_n)$ | `input` | `null`

COMANDI:
c ::= E | $X := E$ | $c_1; c_2$ | `return` E | `output` E | $\underbrace{\ldots}_{\texttt{if,while,for,}ecc.}$

DICHIARAZIONI DI VARIABILI:
V ::= `vars` $X_1 : T_1, \ldots, X_n : T_n$

DICHIARAZIONI DI METODI:
M ::= `method` $m(X_1 : T_1, \ldots, X_n : T_n) : T\{c\}$

DICHIARAZIONI DI CLASSI:
C ::= `class` $A\{V; M_1; \ldots; M_n\}$

PROGRAMMI:
P ::= $C_1 \ldots C_n E$

Esempio 32 Consideriamo una semplice classe per la gestione dell'I/O:

```
class IO {
    vars;
    method get() : int {return input};
    method put(dato : int) : void {output dato};
}
new(IO).put(3)
```

Per la traduzione nel π-calcolo usiamo un π-calcolo poliadico in cui, oltre a nomi, è possibile passare interi. Implementiamo gli oggetti nel modo seguente.

- ogni oggetto è identificato in maniera univoca tramite un nome fresh;
- una volta creato l'oggetto è sempre in ascolto su tale nome;
- i messaggi che un oggetto riceve ed invia sono sequenze di nomi ed interi: il primo è il nome del metodo invocato mentre l'ultimo è un nome fresh generato da chi ha invocato il metodo per avere l'eventuale risultato della chiamata.

Similmente, una classe è:

- associata ad un nome univoco;
- implementata da un processo replicato che crea un nome nuovo e lo spedisce sul nome associato alla classe.

Nella traduzione abbiamo due nomi, *in* e *out*, che denotano canali riservati corrispondenti allo standard input ed output.

- n_{IO} è il nome usato per riferire la classe IO
- La classe viene tradotta come

$$C_{IO} \triangleq !(\nu i)\overline{n_{IO}}\langle i\rangle.O^i_{IO}$$

- Una istanza della classe è tradotta come

$$O^i_{IO} \triangleq i(x,y,z).([x = get]in(w).\overline{z}\langle w\rangle.O^i_{IO} \mid [x = put]\overline{out}\langle y\rangle.O^i_{IO})$$

  Si noti che, in questo caso, sarebbe bastato modellare O^i_{IO} come un processo replicato; ma nel caso generale in cui ci sono variabili di istanza che possono essere modificate tramite l'invocazione di metodi dell'oggetto (vedi l'esempio successivo), bisogna garantire che le diverse chiamate dei metodi lavorino in mutua esclusione sui dati condivisi, per garantirne l'integrità.

- L'espressione da valutare è

$$E_1 \triangleq n_{IO}(t).\overline{t}\langle put, 3, -\rangle$$

  dove '$-$' indica un nome non importante (infatti *put* non restituisce alcun valore)

- Il programma è

$$P_1 \triangleq (\nu n_{IO})(C_{IO} \mid E_1)$$

Esercizio: dimostrare che $P_1 \approx^c \overline{out}\langle 3\rangle$.

Esempio 33 Consideriamo ora la seguente evoluzione del programma dell'esempio precedente:

```
class IO {
    vars;
    method get() : int {return input};
    method put(dato : int) : void {output dato}
}
class A {
    vars o : ref(IO);
    method go() : void {o := new(IO); new(B).print(o, 3)}
}
classB {
    vars;
    method print(q : ref(IO); k : int) : void {q.put(k)}
}
new(A).go()
```

Come prima, *in* e *out* sono i nomi dello stdin e stdout, mentre n_{IO}, n_A e n_B i nomi per le classi. Le classi C_{IO} e O_{IO} sono tradotte come nell'esempio precedente. Inoltre, abbiamo che

$$\begin{aligned}
C_B &\triangleq\ !(\nu b)\overline{n_B}\langle b\rangle.O_B^b \\
O_B^b &\triangleq\ b(x,y,y',z).[x = print]\overline{y}\langle put, y', -\rangle.O_B^b \\
C_A &\triangleq\ !(\nu a, c_0)\overline{n_A}\langle a\rangle.O_A^a\langle null\rangle \\
O_A^a\langle v\rangle &\triangleq\ a(x,z).[x = go]n_{IO}(w).n_B(w').\overline{w'}\langle print, w, 3, -\rangle.O_A^a\langle w\rangle \\
E_2 &\triangleq\ n_A(t).\bar{t}\langle go, -\rangle \\
P_2 &\triangleq\ (\nu n_{IO} n_A n_B)(C_{IO} \mid C_A \mid C_B \mid E_2)
\end{aligned}$$

dove abbiamo applicato la traduzione generale per gli oggetti di classe A e B, sebbene abbiano un solo metodo. Si noti inoltre l'uso di definizioni ricorsive con parametri per modellare lo stato interno dell'oggetto (in generale, un oggetto con n variabili di istanza può essere modellato con una definizione di processo di arità n).

Esercizio: dimostrare che $P_2 \approx^c \overline{out}\langle 3\rangle$.

Quindi, considerando il processo P_1 dell'Esempio 32, abbiamo che $P_1 \approx^c P_2$, poichè entrambi sono equivalenti al processo $\overline{out}\langle 3\rangle$. In questo modo abbiamo un modo formale per verificare equivalenze tra programmi scritti in un semplice linguaggio ad oggetti.

Bibliografia

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. Precedentemente apparso su *Proc. of the 4th ACM Conf. on Computer and Communications Security*, 1997.

[2] H. P. Barendregt. *The Lambda Calculus*, volume 103 in *Studies in Logic and the Foundations of Mathematics*. North Holland, 1984.

[3] M. Boreale and R. De Nicola. Testing equivalences for mobile processes. *Information and Conputation*, 120:279–303, 1995. Precedentemente apparso su *Proc. of CONCUR '92*, LNCS 630.

[4] G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[5] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR'96*; *LNCS*, volume 1119, pagg. 406–421. Springer, 1996.

[6] D. Gorla. On the Relative Expressive Power of Asynchronous Communication Primitives. In *Proc. of 9th Intern. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'06)*; *LNCS*, volume 3921, pagg. 47–62. Springer, 2006.

[7] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.

[8] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*; *LNCS*, volume 512, pagg. 133–147. Springer, 1991.

[9] D. Lowe, X. Chen, T. Mondor, T. Rus, N. Rynearson, S. Wright, and T. Xu. *BizTalk Server: The Complete Reference*. McGraw Hill, 2001.

[10] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[11] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 in *Series F: Computer and System Sciences*. NATO Advanced Study Institute, 1993.

[12] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992. Una versione preliminare è apparsa come Tech. Rep. ECS-LFCS-89-85/86, Laboratory for Foundations of Computer Science, Univ. di Edimburgo, 1989.

[13] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. Precedentemente apparso su *Proc. of POPL'97*, ACM Press.

[14] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[15] D. Sangiorgi. Bisimulation in higher-order process calculi. *Information and Computation*, 131:141–178, 1996. Precedentemente apparso su *Proc. of PROCOMET'94*, pagg. 207–224.

[16] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In *Proceedings of ICCL '98*; *LNCS*, volume 1686. Springer, 1999.

[18] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1995.

[19] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis CST-126-96, 1996 (apparsa anche come ECS-LFCS-96-345).

[20] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *Proceedings of POPL '01*, pagg. 116–127. ACM, 2001.

[21] N. Yoshida. Graph types for monadic mobile processes. In *Proceedings of FSTTCS '96*; *LNCS*, volume 1180, pagg. 371–386. Springer, 1996.