Appunti di Modelli di Calcolo

Andrea Cimino
Lorenzo Muti
Marco Stronati
*

7 maggio 2012

*Licenza Creative Commons: Attribuzione - Non commerciale - Condividi allo stesso modo 2.5 Italia (CC BY-NC-SA 2.5) -

Indice

Introduzione ix

I. Il linguaggio IMP 1

1. La semantica operazionale di IMP 3
1.1. Regole di inferenza e grammatiche . . . 3
1.1.1. Regole di inferenza . . . 3
1.1.2. Un esempio di sistema di regole di inferenza . . . 4
1.2. Sintassi di IMP . . . 7
1.3. Semantica operazionale di IMP . . . 7
1.3.1. Stato della memoria . . . 7
1.3.2. Sistema di regole . . . 8
1.4. Uguaglianza fra programmi in IMP . . . 11
1.4.1. Dimostrazioni di uguaglianza semplici . . . 11
1.4.2. Dimostrazioni per programmi non completamente specificati . . . 13
1.4.3. Computazioni che non terminano . . . 14
1.4.4. Dimostrazioni di non uguaglianza . . . 16

2. Metodi di Induzione 19
2.1. Principio dell'induzione ben fondata di Noether . . . 19
2.1.1. Relazioni ben fondate . . . 19
2.1.2. L'induzione ben fondata di Noether . . . 22
2.2. Altri metodi di induzione . . . 22
2.2.1. Induzione matematica debole . . . 23
2.2.2. Induzione matematica forte . . . 23
2.2.3. Induzione strutturale . . . 23
2.2.4. Induzione sulle derivazioni . . . 25
2.2.5. Induzione sulle regole . . . 27
2.3. Ricorsione ben fondata (Well founded recursion) . . . 27

3. Ordinamenti parziali e teorema del punto fisso 31
3.1. Basi degli ordinamenti parziali . . . 31
3.1.1. Ordinamenti parziali e totali . . . 31
3.1.2. Diagrammi di Hasse . . . 33
3.1.3. Catene . . . 35
3.1.4. Ordinamenti parziali completi (CPO) . . . 38
3.1.5. Reticoli . . . 40
3.2. Verso la teoria del punto fisso . . . 41
3.2.1. Monotonia e continuità . . . 41
3.2.2. Punto fisso . . . 43
3.2.3. Il teorema di Kleene . . . 43
3.2.4. Il teorema di Tarski . . . 46
3.3. Operatore delle conseguenze immediate . . . 46
3.3.1. L'operatore $\hat{R}$ . . . 46
3.3.2. Punto fisso di $\hat{R}$ . . . 48

4. Semantica denotazionale di IMP **51**
4.1. Semantica denotazionale di IMP . . . 51
4.1.1. Funzione $\mathscr{A}$ . . . 51
4.1.2. Funzione $\mathscr{B}$ . . . 52
4.1.3. Funzione $\mathscr{C}$ . . . 52
4.2. Equivalenza fra semantica operazionale e semantica denotazionale . . . 54
4.2.1. Dimostrazione per $\mathscr{A}$ e $\mathscr{B}$ . . . 55
4.2.2. Dimostrazioni per $\mathscr{C}$ . . . 55
4.2.2.1. Dalla semantica operazionale alla denotazionale . . . 56
4.2.2.2. Dalla semantica denotazionale alla operazionale . . . 57
4.3. Dimostrare proprietà con la semantica denotazionale . . . 59

II. Il linguaggio HOFL **63**

5. Sintassi e semantica operazionale di HOFL **65**
5.1. λ-calcolo . . . 65
5.1.1. λ-calcolo non tipato . . . 65
5.1.2. λ-calcolo semplicemente tipato . . . 66
5.2. HOFL . . . 67
5.2.1. Sintassi . . . 67
5.2.2. Regole di inferenza di tipi . . . 68
5.2.3. Problema della tipizzazione . . . 68
5.2.3.1. Tipizzazione alla Church . . . 68
5.2.3.2. Tipizzazione alla Curry . . . 69
5.2.4. Sostituzione . . . 70
5.3. Semantica operazionale di HOFL . . . 71
5.3.1. Forme Canoniche . . . 72
5.3.2. Regole di inferenza . . . 72

6. Complementi di teoria dei domini e semantica denotazionale di HOFL **77**
6.1. Teoria dei domini . . . 77
6.1.1. $CPO_\perp$ per il tipo *int* . . . 78
6.1.2. $CPO_\perp$ per il tipo $\tau_1 \times \tau_2$ . . . 78
6.1.2.1. Costruzione del $CPO_\perp$ . . . 78
6.1.2.2. L'operatore π . . . 78
6.1.3. $CPO_\perp$ per il tipo $\tau_1 \to \tau_2$ . . . 79
6.1.3.1. Continuità in funzioni del tipo $\tau_1 \to (\tau_2 \times \tau_3)$ . . . 80
6.1.3.2. Continuità in funzioni del tipo $D \times E \to S$ . . . 81
6.1.4. Dimostrazioni di continuità . . . 82
6.1.5. Lifting . . . 83
6.2. Semantica denotazionale di HOFL . . . 84
6.2.1. Costanti . . . 85
6.2.2. Variabili . . . 85
6.2.3. Operazioni binarie . . . 85
6.2.4. Condizionale . . . 86
6.2.5. Coppie . . . 87
6.2.6. Selezionatori di coppia . . . 88
6.2.7. Lambda function . . . 88
6.2.8. Applicazione di funzione . . . 90
6.2.9. Ricorsione . . . 90
6.3. La semantica denotazionale è composizionale . . . 90

7. Equivalenza fra le semantiche di HOFL 93
7.1. Basi del confronto . . . 93
7.2. Uso del lifting . . . 93
7.3. La terminazione operazionale implica la terminazione denotazionale . . . 94
7.3.1. Forma canonica . . . 94
7.3.2. Operazioni binarie . . . 94
7.3.3. Condizionale . . . 95
7.3.4. Selettore da coppia . . . 95
7.3.5. Applicazione di funzione . . . 96
7.3.6. Ricorsione . . . 96
7.3.7. Conclusione . . . 96
7.4. La terminazione denotazionale implica la terminazione operazionale . . . 97
7.5. Conclusioni . . . 97

III. Calcoli di processo 101

8. Il linguaggio CCS 103
8.1. Definizione del linguaggio . . . 104
8.1.1. Sintassi di CCS . . . 105
8.1.2. Semantica Operazionale . . . 106
8.1.3. Semantica astratta del CCS . . . 108
8.2. Relazioni di equivalenza . . . 109
8.2.1. Isomorfismo dei grafi . . . 109
8.2.2. Stringhe o *Trace equivalence* . . . 109
8.3. Bisimilarità . . . 110
8.3.1. Punto fisso . . . 112
8.3.1.1. Il $CPO_\perp$ dei processi . . . 112
8.3.1.2. L'operatore Φ . . . 113
8.4. Congruenza e bisimilarità . . . 116
8.4.0.3. Bisimilarità è una congruenza per il parallelo . . . 116
8.4.0.4. Accenno di dimostrazione di bisimilarità per regola del prefisso . . . 118
8.5. Logica di Hennessy - Milner . . . 118
8.6. μ-calcolo . . . 119
8.7. Weak Observational equivalence . . . 120
8.8. Weak Observational Congruence . . . 121
8.9. Dynamic congruence . . . 122
8.10. Caratterizzazione assiomatica della bisimulazione . . . 123
8.10.1. Teorema di Espansione . . . 123
8.10.2. Assiomi di riduzione . . . 123
8.10.3. Assiomi bisimilarità strong . . . 123

9. Il Π-calcolo 125
9.1. Sintassi di Π-calcolo . . . 126
9.1.1. Sintassi e potenzialità . . . 126
9.1.2. Free names e bound names . . . 126
9.2. Semantica operazionale del Π-calcolo . . . 128
9.2.1. Messaggi . . . 128
9.2.2. Nondeterminismo . . . 129
9.2.3. Matching . . . 129
9.2.4. Parallelismo . . . 129
9.2.5. Restrizioni, apertura e chiusura . . . 130
9.2.6. Replicazione . . . 132

9.3. Semantica astratta di Π-calcolo 133
9.3.1. Nozioni di equivalenza 133
9.3.2. Semantica astratta 133
9.3.2.1. Bisimularità ground 133
9.3.2.2. Bisimilarità non ground 134
9.4. Estensioni del Π-calcolo 135
9.4.1. Π-calcolo asincrono 135
9.4.2. HOPI: High-order Π-calculus 135

IV. Calcoli di processo probabilistici 137

10. Teoria della misura e catene di Markov 139
10.1. La teoria della misura 139
10.1.1. I Sigma-field 139
10.1.2. Costruire un sigma-field 140
10.1.3. Distribuzione probabilistca esponenziale 141
10.2. Processi Stocastici Markoviani 143
10.2.1. Le catene di Markov 143
10.2.2. Catene di Markov a tempo discreto e continuo 144
10.3. DTMC - Discrete-time Markov Chain 144
10.3.1. Relazioni con CCS 144
10.3.2. Definizioni 144
10.3.3. Limite di una catena: catene ergodiche 146
10.4. CTMC - Continuous-time Markov Chain 147
10.4.1. Definizioni 147
10.4.2. Bisimilarità nelle CTMC 148

11. Estensioni delle catene di Markov e PEPA 151
11.1. Azioni e nondeterminismo nelle catene di Markov 151
11.1.1. CTMC con azioni 151
11.1.1.1. Reactive CTMC 151
11.1.1.2. Logica di Larsen-Skou 152
11.1.2. CTMC con nondeterminismo 152
11.1.2.1. Segala Automaton - Roberto Segala 153
11.1.2.2. Simple Segala Automaton 153
11.1.2.3. Altre combinazioni 154
11.1.3. Segala come modello più generale 154
11.1.3.1. Segala vs Generative 154
11.1.3.2. Generative → Segala 154
11.1.3.3. Reactive → Segala 155
11.1.3.4. Simple Segala → Segala 155
11.1.3.5. Simple Segala vs Generative 155
11.2. PEPA - Performance Evaluation Process Algebra 155
11.2.1. Sintassi di PEPA 155
11.2.2. Semantica operazionale di PEPA 155

V. Appendici 157

A. Tabelle riassuntive di sintassi e semantica 159
A.1. IMP 159
A.1.1. Sintassi 159

A.1.2. Semantica operazionale 159
A.1.2.1. Regole per Aexpr $< a, \sigma > \rightarrow n \in \mathbb{N}$ 159
A.1.2.2. Regole per Bexpr $< b, \sigma > \rightarrow v \in \{\mathbf{true}, \mathbf{false}\}$ 159
A.1.2.3. Regole per Com $< c, \sigma > \rightarrow \sigma' \in \Sigma$ 159
A.1.3. Semantica denotazionale 160
A.1.3.1. $\mathscr{A} : Aexpr \rightarrow \Sigma \rightarrow \mathbb{N}$ 160
A.1.3.2. $\mathscr{B} : Bexpr \rightarrow \Sigma \rightarrow \mathbb{B}$ 160
A.1.3.3. $\mathscr{C} : Com \rightarrow \Sigma \rightarrow \Sigma_{\perp}$ 160
A.2. HOFL 161
A.2.1. Sintassi 161
A.2.2. Regole di inferenza dei tipi 161
A.2.3. Semantica operazionale 162
A.2.3.1. Forme canoniche 162
A.2.3.2. Regole di inferenza 162
A.2.4. Semantica denotazionale 162
A.3. CCS 163
A.3.1. Sintassi 163
A.3.2. Semantica operazionale 163
A.3.3. Strong Bisimilarity 163
A.3.4. Weak Observational equivalence 163
A.3.5. Weak Observational Congruence 163
A.3.6. Dynamic congruence 164
A.4. Π-calcolo 164
A.4.1. Sintassi 164
A.5. Semantica operazionale 164
A.5.0.1. Bisimilarità ground 164
A.5.0.2. Bisimilarità non ground 164
A.6. Modelli probabilistici 164
A.6.1. DTMC 164
A.6.2. CTMC 165
A.6.3. Reactive 165
A.6.4. Generative 165
A.6.5. Segala 165
A.6.6. Simple Segala 165
A.7. PEPA 165

B. Esercizi **167**
B.0.1. Esercizi prima parte 167
B.0.2. Esercizi 23 Marzo 2010 172
B.0.3. 2 Febbraio 2000 175
B.0.4. 20 Gennaio 2005 177
B.0.5. Prova scritta del 6 aprile 2005 178
B.0.6. Prova scritta del 21 luglio 2005 180
B.0.7. Prova scritta del 3 Aprile 2007 181
B.0.8. Prova scritta del 31/05/06 182
B.0.9. Prova scritta del 30/04/04 183
B.0.10. Prova scritta del 23/07/07 185
B.0.11. Prova scritta del 9 aprile 2010 186
B.0.12. Correzione esercizi 23/03/2010 190
B.0.12.1. Esercizio 1 190
B.0.12.2. Esercizio 3 190
B.0.13. Esercizio 4 191
B.0.13.1. Esercizio 6 193

B.0.13.2. Esercizio 7 193

VI. Elenchi **195**

Introduzione

Il corso di modelli di calcolo è strutturato in 4 parti:

- Modelli per il calcolo imperativo, che studieremo con il linguaggio IMP
- Modelli per il calcolo funzionale, che studieremo con il linguaggio HOFL
- Modelli per i calcoli di processo, che studieremo con i linguaggi CCS e pi-calcolo
- Modelli per i calcoli di processo probabilistici e stocastici

I primi due modelli saranno *deterministici*, mentre gli altri saranno *non-deterministici*. Vedremo in seguito la differenza.

Descriveremo il mondo che ci interessa attraverso la semantica operazionale e la semantica denotazionale. Entrambi questi strumenti ci forniranno una *funzione di interpretazione* che va dalla sintassi alla semantica dei programmi. Questo ci permetterà ad esempio di controllare se due programmi hanno la stessa semantica (il che equivale a dire che fanno la stessa cosa).

La semantica denotazionale descrive la funzione di interpretazione in maniera esplicita.

Nei Calcoli di Processo probabilistici l'evoluzione del sistema è non-deterministica, ovvero, dato un programma, non si può determinare a priori il suo comportamento. Addirittura, in PI-Calcolo i programmi possono evolvere.

I modelli di calcolo *probabilistico* tendono ad assegnare alle decisioni un valori di stima (quindi non certi). Questi modelli hanno molta applicazione ad esempio in Biologia. Il calcolo *stocastico* in particolare riceve l'input ad intervalli di ampiezza esponenziale, dunque presenta delle proprietà particolari.

Inoltre, una tecnica che si può utilizzare per mostrare l'equivalenza tra due programmi è la bi-simulazione: non ci interessa come sono fatti i programmi, che prendiamo come scatole nere. Ci interessa il loro comportamento, che osserviamo attraverso due "spioncini". In questo modo si puó determinare l'equivalenza tra due programmi osservandone il comportamento.

Parte I.

Il linguaggio IMP

1. La semantica operazionale di IMP

1.1. Regole di inferenza e grammatiche

1.1.1. Regole di inferenza

Le regole di inferenza sono strumenti che ci permetteranno di definire la *semantica operazionale*. Vediamone una definizione formale.

Definizione 1.1 (Sistema di regole di inferenza)
Un sistema di regole di inferenza*, detto anche* logica*, è un insieme di oggetti che determinano le cosiddette* formule ben formate *(fbf). Le regole sono definite come*

$$r = \underbrace{\langle x_1, x_2, \ldots, x_n \rangle}_{premesse} / \underbrace{y}_{conseguenza}$$

Questa notazione significa che se ho già nel mio insieme di formule dimostrate tutte *le formule $x_1, x_2, \ldots, x_n$ posso aggiungervi la formula y.*

Nell'ambito della semantica operazionale, una notazione per esprimere queste regole di inferenza è costituita da una barra verticale, sopra la quale vi sono le premesse di una regola e sotto la quale vi è la conseguenza.

Esempio 1.2 (Regola di inferenza)
La regola

$$\frac{a \in I \quad b \in I \quad a \oplus b = c}{c \in I}$$

significa che, se *a e b sono due elementi appartenenti ad un certo insieme I e l'operatore $\oplus$ applicato a tali elementi (qualunque sia il suo significato) restituisce c,* allora *c appartiene all'insieme I.*

La regola

$$\frac{}{5 \in I}$$

è un assioma, in quanto non ha premesse (ovvero, la formula sotto la linea è vera in qualunque caso).

Dalle regole di inferenza nasce un concetto fondamentale del corso, il concetto di *derivazione*.

Definizione 1.3 (Derivazione)
Una derivazione, dato un insieme di regole R, è un oggetto

$$d \vdash_R y$$

tale che

- $(\emptyset/y) \vdash_R y$ *se* $(\emptyset/y) \in R$ *(concetto di* assioma*)*
- $(\{d_1, \ldots, d_n\rangle /y) \vdash_R y$ *se* $(\{x_1, \ldots, x_n\} /y) \in R$ *e* $d_1 \vdash_R x_1, \ldots, d_n \vdash_R x_n$ *($\{x_1, \ldots, x_n$ devono essere già dimostrate)*

Questa definizione costruisce un albero nel quale come radice abbiamo la proposizione alla quale arriviamo tramite la derivazione, e come foglie abbiamo solamente assiomi. Da questa definizione ne derivano altre due che definiscono cosa sia un teorema.

Definizione 1.4 (Teorema)
$\vdash_R y$ *(y è teorema)* $\Leftrightarrow \exists d : d \vdash_R y$

In pratica un teorema è un'affermazione per la quale esiste una derivazione usando le regole R. Definiamo anche l'insieme dei teoremi di R, banalmente, nella maniera seguente.

Definizione 1.5 (Insieme dei teoremi in R)
$I_R = \{y \ : \vdash_R y\}$

1.1.2. Un esempio di sistema di regole di inferenza

Vediamo un esempio.

Esempio 1.6 (Grammatiche viste come sistemi di regole)
Prendiamo una grammatica "classica" che produce tutte le stringhe di parentesi bilanciate:

`S ::= SS | (S) | λ`

Con λ si indica la stringa vuota. Ogni grammatica si può scrivere come un sistema di regole di inferenza. Vediamole con la notazione vista precedentemente.

$$\frac{s_1 \in L \quad s_2 \in L}{s_1 s_2 \in L} \ (1)$$

Questa regola dice che, prese due stringhe s_1 ed s_2 appartenenti al linguaggio L, anche la stringa $s_1 s_2$ appartiene a tale linguaggio: è la regola corrispondente alla produzione S ::= SS. Allo stesso modo possiamo derivare le regole

$$\frac{s \in L}{(s) \in L} \ (2)$$

$$\frac{}{\lambda \in L} \ (3)$$

che corrispondono rispettivamente alle produzioni S ::= (S) e S ::= λ.

Istanziando queste regole possiamo dimostrare formalmente che una stringa appartiene al linguaggio. Notare che la regola

$$\frac{\texttt{)(} \in L \quad \texttt{((} \in L}{\texttt{)(((} \in L} \ (1)$$

è perfettamente valida, nonostante la stringa `)(((` *non appartenga al linguaggio (non essendo una stringa di parentesi bilanciate): il significato della regola è: "se le stringhe* `)(` *ed* `((` *appartengono al linguaggio, allora anche* `)(((` *vi appartiene", e questa affermazione è vera essendo falsa almeno una delle ipotesi:* `)(` *non è una stringa di parentesi bilanciate.*

Passiamo a dimostrare qualcosa di vero.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\lambda \in L}(3)}{(\lambda) = () \in L}(2)}{(()) \in L}(2) \qquad \cfrac{\cfrac{}{\lambda \in L}(3)}{(\lambda) = () \in L}(2)}{(())() \in L}(1)$$

Si legge dall'alto verso il basso: dal fatto che $\lambda \in L$, che è assioma del nostro linguaggio, deduco che $() \in L$ tramite la seconda regola, quindi riapplico la seconda regola per dedurre che $(()) \in L$ ed infine la prima regola per concatenare le due stringhe.

Diamo adesso un **secondo metodo** *per formalizzare questo linguaggio che non usi le regole di inferenza. Detta a una stringa e a_i l'i–esimo carattere di a, definiamo la quantità*

$$f(a_i) = \begin{cases} 1 & se\ a_i = (\\ -1 & se\ a_i =) \end{cases}$$

Possiamo a questo punto dire (affermazione che dimostreremo in seguito) che una stringa a di lunghezza n appartiene al linguaggio se valgono le seguenti proprietà :

1. $\sum_{i=1}^{m} f(a_i) \geq 0 \qquad m = 0, 1...n$
2. $\sum_{i=1}^{n} f(a_i) = 0$

Per chiarire le idee, si riporta un esempio dei valori che assumono queste sommatorie per la stringa $(())()$*:*

$i =$	0	1	2	3	4	5	6
$a_i =$		(	(	)	)	(	)
$f(a_i) =$		1	1	−1	−1	1	−1
$\sum_{j=1}^{i} f(a_j) =$	0	1	2	1	0	1	0

Abbiamo dunque due metodi differenti per descrivere il nostro linguaggio: il primo è un insieme di regole di inferenza e serve a definire *come costruire una stringa che appartiene al nostro linguaggio, il secondo aiuta a* controllare *che una data stringa appartenga al linguaggio.*

Si ricorda che con L si indica il linguaggio delle parentesi bilanciate. Dimostriamo adesso che i due metodi sono equivalenti. Questo equivale a dire che

$$a \in L \iff \begin{cases} \sum_{i=1}^{m} f(a_i) \geq 0 & m = 0, 1...n \\ \sum_{i=1}^{n} f(a_i) = 0 \end{cases}$$

La dimostrazione si divide in due parti:

$\Longrightarrow$ *le stringhe che rispettano il primo metodo (ovvero, sono prodotte dalla grammatica S) rispettano anche il secondo metodo;*

$\Longleftarrow$ *le stringhe che rispettano il secondo metodo rispettano anche il primo, ovvero la grammatica S genera* tutte *le stringhe del linguaggio.*

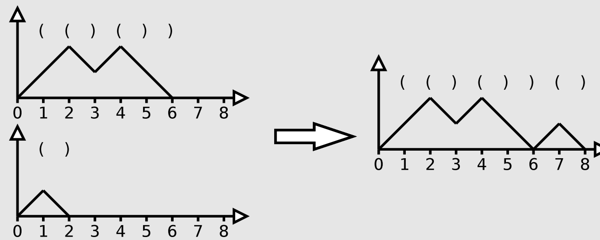
*Dimostriamo $\Longrightarrow$. Come metodo di dimostrazione utilizziamo l'*Induzione sulle regole*: si assume che le premesse siano vere ($a \in L$) e si dimostrano le conseguenze, ovvero:*

Proprietà 1 $\sum_{i=1}^{m} f(a_i) \geq 0 \qquad m = 0, 1...n$ *(Il numero di parentesi aperte è sempre ≥ di quelle chiuse);*

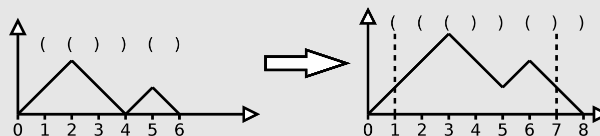
Proprietà 2 $\sum_{i=1}^{n} f(a_i) = 0$ *(Il valore della sommatoria parte da 0 e finisce a 0.*

Considerando il grafico cartesiano che ha come ascisse il valore di i e come ordinate il valore della sommatoria, le due proprietà corrispondono a due proprietà che il grafico deve rispettare: i suoi valori non devono mai andare sotto lo 0 e il grafico deve deve cominciare e finire a 0.

Verifichiamo che applicando ognuna delle tre regole della grammatica, siano ancora rispettate le proprietà 1 e 2. La regola (1) corrisponde al concatenamento dei grafi: date le due stringhe che fanno parte del linguaggio (grafici sulla sinistra), il grafico del concatenamento delle stringhe (grafico a destra) continua a rispettare le proprietà 1 e 2.



La regola (2) corrisponde alla traslazione verso l'alto del grafico a sinistra. Anche in questo caso, le proprietà 1 e 2 sono rispettate.

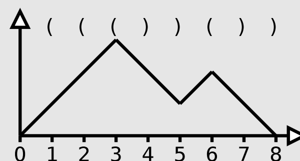


Anche per la regola (3) le proprietà 1 e 2 sono ovviamente rispettate.

Abbiamo dimostrato $\Longrightarrow$. Resta da dimostrare $\Longleftarrow$: le stringhe che rispettano il secondo metodo rispettano anche il primo. Si verifica la proprietà per una qualsiasi stringa che rispetta il secondo metodo. Individuiamo, per una generica stringa, tre possibili casi:

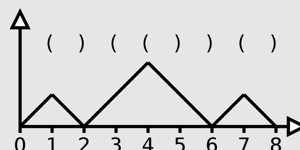
1. *Se $n = 0$, $a = \lambda$. Allora la regola di inferenza (3) ci dice che appartiene al linguaggio ($a \in L$).*

2. *Il grafico inizia da 0, finisce a 0 ma non tocca lo 0. Un esempio è dato dal grafico in figura:*

   In questo caso, si può applicare la regola di inferenza (2) la quale ci dice che data una stringa a', aggiungendo la prima e l'ultima parentesi si ottiene una stringa a lunga $n + 2$ appartenente al linguaggio. Guardando questa regola "al contrario", abbiamo un metodo per verificare che $a \in L$: basta verificare che $a' \in L$.

3. *Il grafico tocca 0 in almeno un punto. Un esempio è dato dal grafico:*

   In questo caso, si utilizza la regola di inferenza (1) e va mostrato che i blocchi 0–2 e 2–8 appartengono al linguaggio.

Abbiamo dimostrato che per ogni possibile stringa del linguaggio, esiste una regola di inferenza. Dunque i due metodi sono equivalenti. □

1.2. Sintassi di IMP

Il linguaggio IMP è una versione ridotta all'osso di C. Ha pochi tipi di dato

- int: numeri interi
- bool: valori booleani
- pointer: puntatori a locazioni di memoria (assumiamo di averne sempre abbastanza)

Non esistono dichiarazioni, né procedure, né blocchi (l'allocazione sarà dunque statica). La grammatica (assolutamente ambigua!) avrà le seguenti produzioni:

Espressioni aritmetiche (Aexpr)

$$a :: n|x|a_0 + a_1|a_0 - a_1|a_0 \times a_1$$

dove n è un naturale, x è una locazione, a_0, a_1 espressioni aritmetiche.

Espressioni booleane (Bexpr)

$$b :: v|a_0 = a_1|a_0 \leq a_1|\neg b|b_0 \vee b_1|b_0 \wedge b_1$$

Comandi (Com)

$$c :: \textbf{skip}\,|x := a|c_0; c_1|\,\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1|\ \textbf{while}\ b\ \textbf{do}\ c$$

Per superare l'ambiguità insita nella grammatica, le espressioni valide dovranno essere sempre parentesizzate.

Esempio 1.7 (Formule valide e non)

$\textbf{while}\ b\ \textbf{do}\ c_1; c_2$	*non è una formula accettabile*
$(\textbf{while}\ b\ \textbf{do}\ c_1); c_2$	*è una formula accettabile*
$\textbf{while}\ b\ \textbf{do}(c_1; c_2)$	*è una formula accettabile*

Data una formula, il controllo che questa sia un teorema nel nostro sistema di regole di inferenza (ovvero una stringa valida del linguaggio) è un processo deterministico.

Esempio 1.8 (Dimostrazione di appartenenza ad IMP)
$\textbf{if}(x = 0)\,\textbf{then}(\textbf{skip})\,\textbf{else}(x := x - 1)) \in Com \longleftarrow$
$x = 0 \in Bexp, \textbf{skip} \in Com, x := (x - 1) \in Com \longleftarrow$
$x \in Aexp, 0 \in Aexp, \textbf{skip} \in Com, x := (x - 1) \in Com \longleftarrow$
$x - 1 \in Aexp \longleftarrow$
$x \in Aexp, 1 \in Aexp \quad \square$

1.3. Semantica operazionale di IMP

1.3.1. Stato della memoria

La macchina che eseguirà IMP avrà una struttura dati $\sigma : \Sigma = Loc \to \mathbb{N}$ che manterrà le associazioni fra l'insieme LOC delle locazioni e i valori in $\mathbb{N}$. LOC sarà sempre grande abbastanza per metterci tutti i valori che desideriamo, e restituirà 0 per tutte le locazioni alle quali non è associato alcun dato.

In definitiva

$$\sigma(x) = \begin{cases} 0 & \text{se x non è definita in } \sigma \\ v & \text{altrimenti} \end{cases}$$

Esiste una memoria chiamata memoria zero, indicata con σ_0. Questa memoria è tale che $\forall x.\sigma_0(x) = 0$.

Indicheremo il contenuto esatto di una memoria con la notazione $\sigma = \left({}^5/_x, {}^{10}/_y\right)$, che indica la memoria che contiene 5 nella locazione identificata da x e 10 nella locazione identificata da y.

Introduciamo infine il concetto di cambio di un valore in memoria: introduciamo l'operatore

$$\sigma[{}^n/_x](y) = \begin{cases} n & \text{se } y = x \\ \sigma(y) & \text{se } y \neq x \end{cases}$$

che cambia il valore restituito da σ per il valore x e lo lascia uguale per tutti gli altri valori. Notare che

$$\sigma[{}^n/_x][{}^m/_x](y) = \begin{cases} m & \text{se } y = x \\ \sigma[{}^n/_x](y) = \sigma(y) & \text{se } y \neq x \end{cases} = \sigma[{}^m/_x](y)$$

la composizione di sostituzioni sulla stessa variabile porta dunque ad una "riscrittura" (allo stesso indirizzo x si assegna prima n e poi m, il risultato è che $\sigma[{}^n/_x][{}^m/_x] = \sigma[{}^m/_x]$).

1.3.2. Sistema di regole

Vogliamo costruire un sistema di regole di inferenza che ci permetta di formalizzare la semantica del nostro linguaggio IMP. Per prima cosa, definiamo la forma che possono avere le formule ben formate del nostro sistema di regole:

Espressione aritmetica $\langle a, \sigma \rangle \to n$
La valutazione di una Aexpr in σ produce come ci possiamo aspettare un numero naturale.

Espressione booleana $\langle b, \sigma \rangle \to u$
La valutazione di una Bexpr in σ produce un valore fra **true** e **false**.

Comando $\langle c, \sigma \rangle \to \sigma'$
La valutazione di un Com in σ produce un nuovo stato σ'.

Passiamo adesso a definire le regole di inferenza che ci permetteranno di comprendere quali di queste formule sono teoremi del nostro sistema.

La regola 1.1 è banale: indica che il valore di una costante numerica n (intesa come entità sintattica) corrisponde proprio al numero n (inteso come entità semantica, ovvero proprio come "idea di numero")

$$\frac{}{\langle n, \sigma \rangle \to n} \; num \tag{1.1}$$

La regola 1.2 è altrettanto banale: la valutazione di un identificatore restituisce il valore associato a quell'identificatore nella memoria σ.

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)} \; ide \tag{1.2}$$

Finalmente una regola con delle premesse: è la regola relativa alla somma. La valutazione della sequenza di simboli $a_0 + a_1$ in σ restituisce un valore corrispondente alla somma matematica (indicata nella regola dal simbolo $\oplus$) dei due valori n_0 e n_1 ottenuti rispettivamente dalla valutazione di a_0 ed a_1 in σ.

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 \oplus n_1} \; sum \tag{1.3}$$

Evitiamo di riportare le formule per le altre operazioni aritmetiche (prodotto, differenza, divisione) e per le operazioni binarie booleane (and e or fra due Bexpr, uguaglianza e confronto fra due Aexpr), che di fatto sono identiche all'operazione riportata qua sopra. Unica piccola variazione è l'operatore di negazione, che è un operatore unario.

$$\frac{\langle b, \sigma \rangle \to v}{\langle \neg b, \sigma \rangle \to \neg v} \; not \tag{1.4}$$

Passiamo alla semantica dei comandi. La regola 1.5 è relativa al comando skip, ed ha una semantica semplicissima: lascia la memoria σ invariata.

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma} \; skip \tag{1.5}$$

La regola 1.6 è relativa all'assegnamento. L'effetto è quello di restituire una memoria identica alla precedente eccetto che per il valore associato ad x.

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[^m/_x]} \; assign \tag{1.6}$$

La regola è relativa alla concatenazione di comandi. Viene valutato il primo comando nella memoria σ per produrre una memoria σ'', nella quale a sua volta verrà valutato il secondo comando. La memoria restituita da questo secondo passaggio è la memoria restituita dai due comandi concatenati.

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'} \; concat \tag{1.7}$$

Passiamo all'**if**. Questo ha bisogno di due regole differenti, in quanto la valutazione di un **if** porta ad uno sdoppiamento del flusso di esecuzione. In particolare, abbiamo la regola 1.8 che mostra il comportamento nel caso in cui la guardia sia vera, e la regola 1.9 nel caso in cui la guardia sia falsa. Il ramo da percorrere sarà deciso **a tempo di esecuzione**.

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1, \sigma \rangle \to \sigma'} \; if - tt \tag{1.8}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1, \sigma \rangle \to \sigma'} \; if - ff \tag{1.9}$$

Vediamo infine la coppia di regole che mostra il comportamento nel caso in cui si trovi un **while**. Anche qua abbiamo uno sdoppiamento: in un caso la guardia è vera, e questo provoca l'esecuzione del corpo del **while** seguito da un'ulteriore valutazione del **while** stesso, mentre nell'altro caso la guardia è falsa e lo stato non viene modificato.

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma'' \rangle \to \sigma'}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma \rangle \to \sigma'} \; while - tt \tag{1.10}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma \rangle \to \sigma} \; while - ff \tag{1.11}$$

Notiamo che il **while** è un comando che ha un comportamento lievemente differente dagli altri: tutte le altre regole tendono a "diminuire" la dimensione delle regole da valutare, scomponendo il comando o l'espressione da valutare in sotto-espressioni. Questo porta intuitivamente a ridurre via via la dimensione delle formule da valutare fino a trovare elementi atomici come variabili o costanti. Il **while** è diverso, in quanto nel suo ramo **true** è definito ricorsivamente. Per questo motivo un **while**, come vedremo, può ciclare all'infinito su se stesso.

Vediamo un esempio di come si possono utilizzare queste regole per la valutazione di un programma.

Esempio 1.9 (Valutazione di un comando tramite la semantica operazionale)
Prendiamo il comando

$$c = \underbrace{x := 0; \underbrace{\textbf{while}\ 0 \le y\ \textbf{do}\ \underbrace{(\underbrace{x := \overbrace{\overbrace{x + (2 \times y)}^{a_1} + 1}^{a}}_{c_3}; \underbrace{y := y - 1}_{c_4})}_{c_2}}_{c_1}}_{c}$$

e vediamo di trovarne la semantica. Procederemo con una tecnica goal–oriented: prendiamo la formula ben formata $\left\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\right\rangle \to \sigma$*, e controlliamo se esiste un* σ *per il quale questa formula è un teorema. è bene notare che questo equivale a chiedersi: "dati una memoria di partenza* $({}^{27}/_{x}, {}^{2}/_{y})$ *e il comando c eseguito sulla memoria, esiste una derivazione in una qualsiasi memoria* σ*?". Notare che questo equivale a chiedersi se il programma termina e, nel caso che termini, quale è il contenuto della memoria* σ *dopo l'esecuzione di c.*

Utilizziamo la notazione "ad albero": in basso manteniamo la formula da dimostrare e sopra costruiamo man mano un albero di regole di transizione. Non riusciamo a farlo però per intero: siamo costretti a lavorare "per pezzi".

Iniziamo con il quadro generale: per valutare l'intero comando c dovremo valutare per prima cosa l'assegnamento, che ci restituirà un ambiente nel quale ad x è stato assegnato 0. Quindi dovremo valutare il while nell'ambiente così generato.

$$\cfrac{\cfrac{\cfrac{}{\langle 0, \sigma\rangle \to 0}\ const}{\langle x := 0, \sigma\rangle \to \sigma\left[{}^{0}/_{x}\right] = \sigma''}\ ass \qquad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c, \sigma\rangle \to \sigma'}\ seq$$

Vediamo la semantica del while. Secondo la regola abbiamo

$$\frac{\langle 0 \le y, \sigma''\rangle \to true \quad \langle c_2, \sigma''\rangle \to \sigma''' \quad \langle c_1, \sigma'''\rangle \to \sigma'}{\langle c_1, \sigma''\rangle \to \sigma'}$$

Ognuna di queste tre formule va a sua volta dimostrata. Ad esempio, vediamo la dimostrazione del confronto.

$$\cfrac{\cfrac{}{\langle 0, \sigma\rangle \to 0}\ const \qquad \cfrac{}{\langle y, \sigma\rangle \to \sigma(y) = 2}\ var \qquad 0 \le 2 = true}{\langle 0 \le y, \sigma''\rangle \to true}\ leq$$

La procedura prevede di continuare a costruire alberi di regole di transizione fino a che sulla cima non si trovano solamente assiomi della nostra teoria.

1.4. Uguaglianza fra programmi in IMP

Due programmi c_1 e c_2 in IMP sono equivalenti se vale la seguente proprietà.

$$c_1 \sim c_2 \iff \forall \sigma, \sigma' (\langle c_1, \sigma \rangle \to \sigma' \Leftrightarrow \langle c_2, \sigma \rangle \to \sigma') \tag{1.12}$$

Questa è la definizione generica, che funziona anche per i programmi che non terminano. Due comandi siano uguali (i programmi sono comandi) se è vero che se uno dei due, terminando, va in un certo stato, allora anche l'altro termina e va nello stesso stato. Se invece uno dei due non termina (ovvero non esiste un σ'), anche l'altro non deve terminare (anche per l'altro non deve esistere un σ'). Quindi due programmi che, dato un generico stato iniziale, non terminano, sono uguali.

A questa definizione può essere affiancata un'altra più operativa, e forse più intuitiva, che però funziona solamente per i programmi che terminano. Per dimostrare che due programmi terminano e sono uguali (ovvero terminano nello stesso stato), si può usare la definizione:

$$\forall \sigma \begin{cases} \langle c_1, \sigma \rangle \to \sigma' \\ \langle c_2, \sigma \rangle \to \sigma' \end{cases} \Longrightarrow c_1 \sim c_2 \tag{1.13}$$

In altre parole, due comandi sono uguali fra di loro se e solo se, per ogni memoria σ di partenza, i due comandi valutati in tale memoria restituiscono come risultato la stessa memoria σ'.

Notare che l'implicazione inversa non è vera: possono esserci dei programmi uguali che non terminano e per i quali dunque non esiste un σ'.

1.4.1. Dimostrazioni di uguaglianza semplici

Le prime dimostrazioni che vedremo saranno dimostrazioni che partono da programmi completamente specificati, che operano su memorie non specificate. Vediamo un primo esempio

Esempio 1.10 (Uguaglianza fra comandi (1))
Vediamo un semplice esempio di come si può dimostrare che i due comandi

$$c_1 = x := 1; x := 0$$
$$c_2 = x := 0$$

sono semanticamente equivalenti. Per farlo basta derivare c_1 e c_2 come segue. Cominciamo da c_1:

1. $\langle c_1, \sigma \rangle \to \sigma' \longleftarrow$
   Si parte dalla formula che vogliamo dimostrare, senza legarsi ad alcuno stato in particolare (σ è la generica memoria).

2. $\langle x := 1, \sigma \rangle \to \sigma'' \quad \langle x := 0, \sigma'' \rangle \to \sigma' \xleftarrow{\sigma''=\sigma[^1/_x]}$
   c_1 è un comando composto da due comandi (di assegnamento) concatenati. Usiamo la relativa regola di inferenza 1.7 (pag. 9) introducendo un σ'' che è lo stato della memoria dopo il primo assegnamento e prima del secondo.

3. $\langle x := 0, \sigma \left[^1/_x\right] \rangle \to \sigma' \xleftarrow{\sigma'=\sigma[^1/_x][^0/_x]=\sigma[^0/_x]}$
   Si esegue il secondo assegnamento semplificando la doppia sostituzione della x (la seconda sostituzione mangia la prima). Abbiamo ottenuto $\sigma' = \sigma[^0/_x]$

4. $\square$

Per quanto riguarda il secondo comando c_2, la derivazione è ancora più semplice:

1. $\langle c_2, \sigma \rangle \to \sigma' \longleftarrow$

2. $\langle x := 0, \sigma \rangle \to \sigma' \xleftarrow{\sigma' = \sigma[^0/_x]}$

3. $\square$

In entrambi i casi otteniamo lo stato risultante $\sigma' = \sigma[^0/_x]$: le semantiche dei due programmi sono dunque equivalenti in accordo con la definizione 1.13.

Esempio 1.11 (Uguaglianza fra comandi (2))
Vogliamo dimostrare che i due comandi

$$\begin{aligned} c_1 &= \textbf{while}\, x \neq 0\, \textbf{do}\, x := 0 \\ c_2 &= x := 0 \end{aligned}$$

sono semanticamente uguali.

Possiamo facilmente dimostrare che $\forall \sigma\ \langle c_2, \sigma \rangle \to \sigma'$ con $\sigma' = \sigma[0/x]$, in quanto questo è il significato dell'assegnamento. Abbiamo dunque come risultato una memoria che si comporta ovunque come σ tranne che in x, dove restituisce sempre 0. In particolare, se $\sigma(x) = 0$ abbiamo che $\sigma' = \sigma$.

*Passiamo ad eseguire simbolicamente c_1. Per farlo, è necessario considerare entrambe le regole di inferenza del **while** (1.11 e 1.10 a pag. 9) in quanto non sappiamo quanto valga x, e di conseguenza se la guardia del **while** valga **true** o **false**, nella memoria σ.*

Consideriamo per primo il caso in cui $\sigma(x) \neq 0$. In questo caso abbiamo

$$\begin{aligned} &\langle c_1, \sigma \rangle \to \sigma' \longleftarrow \\ &\sigma(x) \neq 0 \quad \langle x := 0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma' \xleftarrow{\sigma'' = \sigma[^0/_x]} \\ &\sigma(x) \neq 0 \quad \sigma\left[^0/_x\right](x) = 0 \quad \langle c_1, \sigma\left[^0/_x\right] \rangle \to \sigma' \xleftarrow{\sigma' = \sigma[^0/_x]} \square \end{aligned}$$

*Il comando viene valutato seguendo il ramo **true**, quindi viene sviluppato eseguendo il comando x:=0 e valutando nuovamente il **while** nello stato così modificato. Evidentemente nel nuovo stato avrò che x sarà uguale a 0, quindi il **while** terminerà restituendo una memoria identica a σ eccetto che per il valore di x.*

Più semplice è il caso in cui $\sigma(x) = 0$: infatti

$$\begin{aligned} &\langle c_1, \sigma \rangle \to \sigma' \longleftarrow \\ &\sigma(x) = 0 \xleftarrow{\sigma' = \sigma} \square \end{aligned}$$

Per concludere, si osserva che:

- *Se $\sigma(x) = 0$, allora* $\begin{cases} \langle c_1, \sigma \rangle \to \sigma \\ \langle c_2, \sigma \rangle \to \sigma[^0/_x] = \sigma \end{cases}$

- *Se invece $\sigma(x) \neq 0$, allora* $\begin{cases} \langle c_1, \sigma \rangle \to \sigma[^0/_x] \\ \langle c_2, \sigma \rangle \to \sigma[^0/_x] \end{cases}$

Quindi per tutti i valori di x i due comandi applicati alla generica memoria σ producono la stessa memoria. In accordo con la definizione 1.13, i due programmi sono equivalenti.

Abbiamo dunque capito l'idea di base. Il metodo di dimostrazione per computazioni terminanti è sempre il solito: si "sviluppano" le formule per i due comandi e si mostra che la memoria risultatante è la stessa.

1.4.2. Dimostrazioni per programmi non completamente specificati

Fino ad ora abbiamo visto dimostrazioni riguardanti programmi completamente specificati: sapevamo in dettaglio quali istruzioni erano eseguite, e lasciavamo parametrica solamente la memoria σ. Passiamo adesso ad un insieme di esempi nei quali lavoriamo su programmi parametrici: nel prossimo esempio vedremo un uguaglianza fra un **while** ed un **if** opportunamente formulato che vale *per qualunque guardia b e per qualunque corpo c*.

Questo non ci porterà particolari complicazioni: basterà lasciare parametrici i risultati della valutazione di tali blocchi e dimostrare che le nostre conclusioni valgono qualunque sia il loro contenuto.

Esempio 1.12 (Dimostrazioni parametriche (1))
Prendiamo i due comandi

$$\begin{aligned} c_1 &= \textbf{while}\, b\, \textbf{do}\, c \\ c_2 &= \textbf{if}\, b\, \textbf{then}\, c; \textbf{while}\, b\, \textbf{do}\, c\, \textbf{else skip} \end{aligned}$$

è vero che $\forall b, c \quad c_1 \sim c_2$*?*

Analizziamo l'esecuzione di c_1 *nella generica memoria di partenza* σ*. Per analizzare il* ***while****, non potendo valutare con esattezza la sua guardia, dobbiamo necessariamente analizzare i due casi in cui la guardia valga* ***true*** *e in cui la guardia valga* ***false****.*

Cominciamo supponendo che $\langle b, \sigma\rangle \to \textbf{false}$*: ne segue che*

$$\begin{aligned} &\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma\rangle \to \textbf{false} \xleftarrow{\sigma=\sigma'} \square \end{aligned}$$

Dunque se la guardia da ***false****, il risultato, come ci potevamo aspettare, è che lo stato non viene cambiato. Cosa accade invece quando si esegue il secondo comando* c_2 *sulla stessa memoria* σ*? Vediamolo (con* w *si indica il codice del* ***while*** *contenuto in* c_2*).*

$$\begin{aligned} &\langle \textbf{if}\, b\, \textbf{then}\, c; w\, \textbf{else skip;}, \sigma\rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma\rangle \to \textbf{false} \quad \langle \textbf{skip}, \sigma\rangle \to \sigma' \xleftarrow{\sigma'=\sigma} \square \end{aligned}$$

Anche in questo caso otteniamo lo stesso comportamento: la valutazione del comando non porta alcun cambiamento alla memoria. Quindi se $\langle b, \sigma\rangle \to \textbf{false}$ *allora, in accordo con la definizione 1.13,* $c_1 \sim c_2$*.*

Passiamo a valutare cosa accade nel caso in cui $\langle b, \sigma\rangle \to \textbf{true}$*. Per* c_1 *abbiamo il seguente risultato.*

$$\begin{aligned} &\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma' \end{aligned}$$

Fermiamoci qua, in quanto a questo punto per continuare dovremmo fare altre ipotesi sul risultato della guardia del ***while****. Proviamo a sviluppare* c_2*.*

$$\begin{aligned} &\langle \textbf{if}\, b\, \textbf{then}\, c; \textbf{while}\, b\, \textbf{do}\, c\, \textbf{else skip}, \sigma\rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma\rangle \to \textbf{true} \quad \langle c; \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma' \end{aligned}$$

Anche in questo caso, la valutazione di c *in* σ *porta ad un nuovo stato* σ''*, nel quale viene valutato il comando* **while** b **do** c *producendo lo stato* σ' *risultante. Come si vede dall'ultima riga degli sviluppi di* c_1 *e* c_2*, questi si riducono agli stessi sotto–goal, quindi hanno la stessa semantica.*

Se due sviluppi si riducono agli stessi sotto–goal, significa che

$$\forall \sigma, \sigma' (\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma' \Leftrightarrow \langle \textbf{if}\, b\, \textbf{then}\, c; \textbf{while}\, b\, \textbf{do}\, c\, \textbf{else skip}, \sigma\rangle \to \sigma')$$

ovvero, in accordo con la definizione 1.12, abbiamo dimostrato che con $\langle b, \sigma\rangle \to \textbf{true}$*,* $c_1 \sim c_2$*. Unendo i due risultati, si ha che è sempre* $c_1 \sim c_2$*.* $\square$

Vale la pena di notare che quello appena presentato (riduzione agli stessi sotto–goal) è in alcuni casi l'unico metodo che si ha per valutare la semantica dei comandi **while**. Essendo il **while** l'unico comando di IMP che può ciclare all'infinito, nei casi in cui la guardia del **while** sia sempre **true** oppure non sia data, anche il suo sviluppo in sotto–goal è infinito. Ad esempio, nel caso analizzato adesso non si può sapere se il comando terminerà oppure no (dipende da c, b e σ), quello che abbiamo dimostrato è che c_1 termina se e solo se termina c_2, e che se terminano lo stato finale è uguale.

Notare inoltre che stavolta, proprio a causa della possibilità di non–terminazione, per dimostrare $c_1 \sim c_2$ con $\langle b, \sigma \rangle \to \textbf{true}$ abbiamo dovuto utilizzare la definizione di ugualianza 1.12 invece della 1.13.

Per chiarire, vediamo una leggera variazione sull'esempio appena visto.

Esempio 1.13 (Dimostrazioni parametriche (2))
Prendiamo i due comandi

$$\begin{aligned} c_1 &= \textbf{while}\, b\, \textbf{do}\, c \\ c_2 &= \textbf{if}\, b\, \textbf{then}\, w\, \textbf{else skip} \end{aligned}$$

è vero che $c_1 \sim c_2$?

Visto che c_1 è identico al caso precedente, e che lo stesso accade per il ramo di c_2 nel quale $\langle b, \sigma \rangle \to$ **false***, ci limitiamo ad analizzare quanto accade per c_2 nel caso in cui $\langle b, \sigma \rangle \to$* **true***. Banalmente in questo caso abbiamo*

$$\begin{aligned} &\langle \textbf{if}\, b\, \textbf{then while}\, b\, \textbf{do}\, c\, \textbf{else skip}, \sigma \rangle \to \sigma' \longleftarrow \\ &\langle b, \sigma \rangle \to \textbf{true} \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma \rangle \to \sigma' \end{aligned}$$

Ci riconduciamo, in definitiva, all'altro caso: dobbiamo valutare un comando **while** *b* **do** *c in un ambiente σ nel quale la valutazione di un'espressione booleana b restituisce* **true***. Anche questi due comandi sono uguali.*

1.4.3. Computazioni che non terminano

Fino ad ora abbiamo avuto vita facile: abbiamo valutato programmi parametrici o con dati che non mandavano il programma in loop. Ma cosa accade quando il programma è specificato ed il suo testo può provocare situazioni di loop? In questi casi dobbiamo riuscire ad indicare le memorie σ per le quali questo accade.

Riscrivendo per renderla più semplice la definizione di equivalenza tra programmi 1.12, si vede che c_1 e c_2 sono equivalenti se

$$\forall \sigma, \sigma' \quad \bigcup \begin{cases} \langle c_1, \sigma \rangle \to \sigma' \Leftrightarrow \langle c_2, \sigma \rangle \to \sigma' \\ \langle c_1, \sigma \rangle \nrightarrow \Leftrightarrow \langle c_2, \sigma \rangle \nrightarrow \end{cases}$$

$c_1 \sim c_2$ se, dato un σ di partenza, terminano nello stesso stato oppure entrambi non terminano.

Passiamo a vedere alcuni esempi nei quali questa definizione viene messa in pratica.

Esempio 1.14 (Dimostrazioni di non–terminazione)
Prendiamo i due comandi

$$\begin{aligned} c_1 &= \textbf{while}\, x > 0\, \textbf{do}\, x := 1 \\ c_2 &= \textbf{while}\, x > 0\, \textbf{do}\, x := x + 1 \end{aligned}$$

è vero che $c_1 \sim c_2$?

Come sempre, possiamo studiare il comportamento di c_1 dividendo lo sviluppo in due rami, uno nel quale $\sigma(x) \le 0$ e l'altro nel quale $\sigma(x) > 0$. Nel primo caso vale

$$\begin{aligned} &\langle c_1, \sigma \rangle \to \sigma' \longleftarrow \\ &\sigma(x) \le 0 \xleftarrow{\sigma' = \sigma} \square \end{aligned}$$

Il corpo del **while** *non viene eseguito, e il comando restituisce la stessa identica memoria di partenza. Svolgendo in maniera simile per c_2, si ottiene che c_1 e c_2 vanno nello stesso stato σ di partenza.*

Passiamo alla valutazione di c_1 nel caso $\sigma(x) > 0$ in cui il corpo del **while** *viene eseguito: qua ci aspetta una sorpresina.*

$$\begin{array}{l}
\langle c_1, \sigma \rangle \to \sigma' \longleftarrow \\
\sigma(x) > 0 \quad \langle x := 1, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma' \xleftarrow{\sigma'' = \sigma[{}^1/_x]} \\
\sigma(x) > 0 \quad \langle c_1, \sigma \left[{}^1/_x\right] \rangle \to \sigma' \longleftarrow \\
\sigma(x) > 0 \quad \langle x := 1, \sigma \left[{}^1/_x\right] \rangle \to \sigma''' \quad \langle c_1, \sigma''' \rangle \to \sigma' \xleftarrow{\sigma''' = \sigma[{}^1/_x][{}^1/_x] = \sigma[{}^1/_x]} \\
\quad \vdots
\end{array}$$

Otteniamo ciclicamente lo stesso identico stato. Questo può significare solamente una cosa: essendo il processo di derivazione deterministico, lo svolgimento è destinato a svilupparsi in infiniti passaggi, in quanto immancabilmente continueremo a trovare lo stesso identico stato senza riuscire mai a ridure i passaggi a qualcosa di più piccolo.

Questo problema si presenta soltanto con il **while** *perché è questo l'unico comando definito per ricorsione strutturale (vedi la definizione 1.10 a pagina 9), e che di conseguenza può non–terminare.*

Vediamo cosa accade per c_2 nel caso $\sigma(x) > 0$.

$$\begin{array}{l}
\langle c_2, \sigma \rangle \to \sigma' \longleftarrow \\
\sigma(x) > 0 \quad \langle x := x + 1, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma' \xleftarrow{\sigma'' = \sigma[{}^{\sigma(x)+1}/_x]} \\
\sigma(x) > 0 \quad \langle c_1, \sigma \left[{}^{\sigma(x)+1}/_x\right] \rangle \to \sigma' \longleftarrow \\
\sigma(x) > 0 \quad \langle x := x + 1, \sigma \left[{}^{\sigma(x)+1}/_x\right] \rangle \to \sigma''' \quad \langle c_1, \sigma''' \rangle \to \sigma' \\
\qquad \xleftarrow{\sigma''' = \sigma[{}^{\sigma(x)+1}/_x]\left[{}^{\sigma[{}^{\sigma(x)+1}/_x](x)+1}/_x\right] = \sigma[{}^{\sigma(x)+2}/_x]} \\
\qquad \vdots
\end{array}$$

Si raggiunge la stessa situazione ottenuta in precedenza. Intuitivamente possiamo dire che il nostro **while** *è in loop, ma formalmente non possiamo dirlo con immediatezza in quanto ad ogni passo lavoriamo su una memoria diversa.*

Tuttavia, è possibile dimostrarlo con una tecnica induttiva, anticipando un po' quel che vedremo nel prossimo capitolo: per induzione sul numero di iterazioni (vedere sezione 2.2.1) abbiamo che

- *al passo 0, cioè alla prima iterazione, il* **while** *non termina*
- *se all'i-esimo passo il* **while** *non è terminato, non lo sarà neppure all'$(i + 1)$-esimo, in quanto $x > 0 \Rightarrow x + 1 > 0$.*

Una dimostrazione formale richiederebbe di dimostrare che alla k-esima iterazione x varrà $\sigma(x) + k$, e che dunque x non può essere minore di 0. Il while dunque non terminerà mai per $\sigma(x) > 0$ iniziale, come nel caso precedente. Dunque, secondo la definizione 1.12, $c_1 \sim c_2$.

Consideriamo il comando $w = \textbf{while}\, b\, \textbf{do}\, c$. Come abbiamo appena visto, per dimostrare la non terminazione del comando, possiamo utilizzare la regola di inferenza sullo stato delle memorie, applicando l'induzione matematica:

$$\frac{\langle c, \sigma_i \rangle \to \sigma_{i+1} \quad i = 0, 1, \ldots m \quad \forall \sigma_i . \langle b, \sigma_i \rangle \to \textbf{true}}{\langle w, \sigma_0 \rangle \nrightarrow} \tag{1.14}$$

$\sigma(i)$ è la memoria all'inizio della i-esima esecuzione di c. Se in tutte le memorie dei passi successivi a 0 la guardia è valutata in **true**, il **while** non termina.

In alcuni casi, può essere più semplice l'applicazione del seguente risultato, che deriva dalla formula sopra:

$$\frac{\forall \sigma \in S.\langle c, \sigma\rangle \to \sigma' \quad \sigma' \in S \quad \forall \sigma \in S.\langle b, \sigma\rangle \to \mathbf{true}}{\langle w, \sigma\rangle \nrightarrow} \tag{1.15}$$

Se esiste un insieme S di memorie tali che, per ognuna delle memorie in S, la valutazione della guardia b in quella memoria risulta **true** e la valutazione del corpo c in quella memoria produce una memoria anch'essa appartenente ad S, allora w valutato in una qualunque memoria di S non termina.

A livello intuitivo, questo significa che se la valutazione di c fa ricadere lo stato sempre nel solito insieme finito senza mai uscirne, allora il comando non termina.

Da notare che questa definizione è più restrittiva della generale 1.14, dato che funziona solo se il ciclo genera un numero finito di stati, ma può essere di più facile applicazione. Ad esempio, facendo riferimento all'Esempio 1.14, la non-terminazione di c_1 si può dimostrare con 1.15, mentre la non-terminazione di c_2 si dimostra con 1.14.

1.4.4. Dimostrazioni di non uguaglianza

Esempio 1.15 (Dimostrazioni di non–uguaglianza)
Dimostriamo che

$$(\mathbf{while}\, x > 0\, \mathbf{do}\, x := 1); x := 0 \quad \nsim \quad x := 0$$

*Guardiamo per ora soltanto la prima parte del primo comando (il **while**). Nel caso $\sigma(x) \leq 0$ si dimostra facilmente che*

$$\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma\rangle \to \sigma$$

Vediamo la dimostrazione.

$$\cfrac{\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma\rangle \to \sigma}{\cfrac{\langle x > 0, \sigma\rangle \to v = \mathbf{false}}{\cfrac{\langle x, \sigma\rangle \to n}{n = \sigma(x)} \quad \cfrac{\langle 0, \sigma\rangle \to m}{m = 0} \quad n > m}}$$

Invece con $\sigma(x) > 0$

$$\cfrac{\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma\rangle \to \sigma'}{\cfrac{\langle x > 0, \sigma\rangle \to v = \mathbf{true}}{\cdots} \quad \cfrac{\langle x := 1, \sigma\rangle \to \sigma''}{\sigma'' = \sigma[1/x]} \quad \cfrac{\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma''\rangle \to \sigma'}{\cdots \quad \langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma[1/x]\rangle \to \sigma'}}$$

Espandiamo il sotto–goal più in basso a destra.

$$\cfrac{\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma[1/x]\rangle \to \sigma'}{\cfrac{\langle x > 0, \sigma[1/x]\rangle \to v' = \mathbf{true}}{\cdots} \quad \cfrac{\langle x := 1, \sigma[1/x]\rangle \to \sigma[1/x]}{\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma[1/x]\rangle \to \sigma'}}$$

Otteniamo lo stesso sotto–goal: $\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma[1/x]\rangle \to \sigma'$, quindi non esiste alcuna prova, dato che il sotto–goal individuato dovrebbe essere una sottoprova di se stesso, e questo è assurdo.

La conclusione è che

$$\forall \sigma, \sigma'.\langle \mathbf{while}\, x > 0\, \mathbf{do}\, x := 1, \sigma\rangle \to \sigma' \Rightarrow \sigma'(x) > 0 \wedge v' = v$$

Valutiamo a questo punto il primo comando completo.

$$\cfrac{\langle (\mathbf{while}\, x > 0\, \mathbf{do}\, x := 1); x := 0, \sigma\rangle \to \sigma'}{\langle (\mathbf{while}\, x > 0\, \mathbf{do}\, x := 1), \sigma\rangle \to \sigma'' \quad \langle x := 0, \sigma''\rangle \to \sigma'}$$

Nel caso $\sigma(x) > 0$, per quanto visto prima, non c'è alcuna prova.
Guardiamo invece il caso $\sigma(x) \leq 0$

$$\cfrac{\langle(\textbf{while}\ x > 0\ \textbf{do}\ x := 1); x := 0, \sigma\rangle \to \sigma'}{\cfrac{\langle(\textbf{while}\ x > 0\ \textbf{do}\ x := 1), \sigma\rangle \to \sigma}{\vdots \atop \sigma(x) \leq 0} \quad \cfrac{\langle x := 0, \sigma\rangle \to \sigma'}{\vdots \atop \sigma' = \sigma[0/x]}}$$

Il comando termina con $\sigma' = \sigma[0/x]$. Quindi, confrontando il primo ed il secondo comando, si può dire che

$$\exists \sigma, \sigma'. \begin{cases} \langle x := 0, \sigma\rangle \to \sigma' \\ \langle(\textbf{while}\ x > 0\ \textbf{do}\ x := 1); x := 0, \sigma\rangle \nrightarrow \end{cases}$$

Ovvero, esiste una coppia (σ, σ') per la quale un comando termina e l'altro no. Un esempio è costituito da ogni σ con $\sigma(x) = 1$ e $\sigma' = \sigma[0/x]$. Dunque, in accordo con il teorema 1.12 a pagina 11, i comandi non sono uguali.

è bene notare un'accortezza logica che si è usata per dire che i comandi non sono uguali a partire dall'affermazione sopra. Per contraddire un'espressione logica $\forall$ come quella che compare in 1.12, è sufficiente l'esistenza di un elemento che non rispetta la clausola, detto in termini matematici:

$$\neg\forall x.P(x) \Leftrightarrow Q(x) = \exists x.(P(x) \wedge \neg Q(x)) \vee (\neg P(x) \wedge Q(x))$$

Esempio 1.16 (Estensione della grammatica di Aexpr)
è possibile rivedere la regola per il prodotto aritmetico definita in 1.3 ottenendo dei miglioramenti. Guardiamo innanzi tutto la regola come era stata definita originariamente:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}$$

Una nuova coppia di regole equivalente dal punto di vista semantico è la seguente:

$$\frac{\langle a_0, \sigma\rangle \to 0}{\langle a_0 \times a_1, \sigma\rangle \to 0} \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad n_0 \neq 0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}$$

In sostanza, la nuova coppia di regole permette, per le moltiplicazioni il cui primo fattore è 0, di evitare il calcolo del secondo fattore (che è inutile, visto che $0 \cdot n = 0\ \forall n$). Infatti, se si dovesse valutare un'espressione di questo tipo, sarebbe sufficiente l'uso della prima regola, mentre per tutti gli altri casi si potrebbe usare la regola di sempre.

Dimostriamo a questo punto che la regola vecchia e quelle nuove sono uguali dal punto di vista semantico e che entrambe sono deerministiche. Si dimostra per Induzione stutturale:

$P(a) \stackrel{\text{def}}{=} \langle a, \sigma\rangle \to n \quad \langle a, \sigma\rangle \to n' \Rightarrow n = n'$

$P(a_0 \times a_1) \stackrel{\text{def}}{=} \langle a_0 \times a_1, \sigma\rangle \to n \quad \langle a_0 \times a_1, \sigma\rangle \to n' \Leftrightarrow n = n' \stackrel{?}{=}$

4 casi: Due modi possibili per ridurre: 2 li abbiamo gia visti nella dim precedente precedente.

$$\frac{\langle a_0 \times a_1, \sigma\rangle \to n}{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}\ n = n_1 \times n_0$$

$$\frac{\langle a_0 \times a_1, \sigma\rangle \to n'}{\langle a_0, \sigma\rangle \to 0}\ n = 0$$

Per ipotesi induttiva $n_0 = 0$ (Assumiamo la proprietà per le sotto espressioni, induzione strutturale).

2. Metodi di Induzione

In questo capitolo studieremo i metodi di induzione che utilizzeremo in seguito per dimostrare alcune proprietà dei linguaggi che analizzeremo.

Esistono molti tipi di metodi di induzione, ma tutti si basano sul principio di *induzione ben fondata di Noether*. Studieremo dunque prima questo principio, per poi derivare gli altri metodi successivamente.

2.1. Principio dell'induzione ben fondata di Noether

2.1.1. Relazioni ben fondate

Partiamo da alcune definizioni che ci serviranno in futuro per definire il concetto di relazione ben fondata. Ricordiamo innanzitutto la definizione di relazione.

Definizione 2.1 (Relazione su un insieme A)
Una relazione $\prec$ su un insieme A è un sottoinsieme del prodotto cartesiano $A \times A$

$$\prec \subseteq A \times A$$

Una volta definito questo concetto, possiamo definire la catena discendente infinita.

Definizione 2.2 (Catena discendente infinita)
Data una relazione $\prec$ su un insieme A, una catena discendente infinita è una successione infinita di elementi di A tali che

$$\{a_i\}_{i \in \omega} \text{ tale che } \forall i \in \omega . a_{i+1} \prec a_i$$

è una funzione che va da ω in A nella quale gli elementi sono via via sempre più piccoli all'interno dell'ordinamento ($a_0 \prec a_1 \prec a_2$).

Possiamo dunque definire la nostra relazione ben fondata.

Definizione 2.3 (Relazione ben fondata)
Una relazione è ben fondata se non esiste alcuna catena discendente infinita relativa a tale relazione.

Come sempre, definiamo la chiusura transitiva della relazione.

Definizione 2.4 (Chiusura transitiva)
Data una relazione $\prec$, la sua chiusura transitiva $\prec^+$ è il minimo insieme tale che

- $a \prec^+ b \rightarrow a \prec b$
- $a \prec^+ b \land b \prec c \rightarrow a \prec^+ c$

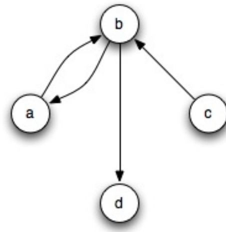


Figura 2.1.: Rappresentazione di una relazione come grafo

Passiamo adesso a dimostrare proprietà su queste relazioni

Teorema 2.5 (Proprietà delle relazioni ben fondate)
Se una relazione $\prec$ è ben fondata, allora è anche senza cicli

$$\forall x | x \not\prec^{+} x \quad \stackrel{\text{def}}{=} \quad \prec \ \textit{non ha cicli}$$

Dimostrazione. Dimostrazione banale: se x ha cicli posso costruire la sequenza

$$x \prec x_1 \prec \ldots \prec x_n \prec x \prec x_1 \prec \ldots \prec x_n \prec x \prec x_1 \prec \ldots$$

che è evidentemente infinita. ☠

Vale anche il seguente

Teorema 2.6 (Relazione ben fondata su insiemi finiti)
Se una relazione $\prec$ su un insieme A è senza cicli e A è un insieme finito, allora $\prec$ è ben fondata.

Dimostrazione. Anche questa dimostrazione è piuttosto banale: se A è finito, una sequenza discendente di elementi lunga $||A|| + 1$ deve, per il *pigeon hole theorem*, avere due elementi uguali che costituirebbero un ciclo. ☠

Possiamo rappresentare queste relazioni come grafi orientati come quello in figura 2.1, nei quali i nodi sono elementi dell'insieme e un arco dal nodo n al nodo m rappresenta la presenza della coppia (n,m) nella relazione.
Vediamo un lemma che ci permetterà di dimostrare in seguito il principio di induzione di Noether.

Lemma 2.7 (Relazione ben fondata)
Data una relazione $\prec$ su un insieme a, questa è ben fondata se e solo se ogni sottoinsieme non vuoto Q di A possiede un elemento m tale che $\forall b \prec m.\ b \notin Q$

Dimostrazione. Come sempre, per dimostrare la doppia implicazione dimostriamo separatamente i due sensi dell'implicazione stessa.

$\Leftarrow$ Supponiamo che ogni sottoinsieme non vuoto di A abbia un elemento minimale. Supponiamo per assurdo che esista una catena discendente infinita: i suoi elementi formerebbero un sottoinsieme di A che non avrebbe un elemento minimale (nel qual caso la catena si fermerebbe a tale elemento). Questo rompe l'ipotesi, quindi ne dobbiamo concludere che non esistano catene discendenti infinite.

⇒ Prendiamo un qualunque sottoinsieme non vuoto Q di A. Prendiamo un elemento $a_0 \in Q$ in modo casuale, e assumiamo che esista una catena $a_n \prec \ldots \prec a_0$. Allora si hanno due possibilità
  - non esiste alcun $b \prec a_n$ in Q: in questo caso a_n è ovviamente un elemento minimale.
  - esiste un $b \prec a_n$ in Q: poniamo $b = a_{n+1}$ ed aggiungiamolo alla catena, per poi ripetere lo stesso controllo. Essendo $\prec$ ben fondata, troveremo sicuramente un elemento a_m in Q tale che non esiste alcun $b \prec a_m$ in Q. Tale elemento è l'elemento minimale in Q.

Avendo dimostrato entrambe le implicazioni, la doppia implicazione è dimostrata.

Ricordiamo la differenza tra elemento minimo e minimale:

- il minimo di un insieme è l'elemento più piccolo di ogni altro, cioè è in relazione con ogni altro elemento dell'insieme
- un elemento minimale di un insieme non ha elementi più piccoli all'interno dell'insieme, tuttavia possono esistere altri elementi minimali che quindi non sono in relazione tra loro.

Vediamo adesso alcuni esempi di relazioni infinite ben fondate e non.

Esempio 2.8 (Numeri Naturali)
Il minore (stretto) aritmetico sull'insieme $\mathbb{N}$ dei numeri naturali è un esempio di relazione ben fondata. Infatti

- *non ha ovviamente cicli*
- *ha un elemento minore rispetto a tutti gli altri, che è lo zero. Qualunque elemento $n \in \mathbb{N}$ si prenda, da esso si può far partire una catena discendente lunga al massimo n*

Esempio 2.9 (Termini e sottotermini)
Possiamo definire i nostri programmi come termini di una segnatura Σ con più sort S e definire una relazione ben fondata che leghi questi termini con i loro sottotermini

$$t_i \prec \sigma(t_1, \ldots, t_n) \qquad i = 1, \ldots, n$$

In pratica, dato un termine t_i questo è minore di tutti i termini che lo utilizzano come argomento. Ad esempio con una segnatura $\Sigma_0 = \{c\}$ $\Sigma_2 = \{f\}$ abbiamo che:

$$c \prec f(c,c) \prec f(f(c,c),c) \prec f(f(f(c,c),c), f(c,c))$$

Anche qua abbiamo un elemento minimo che chiude qualunque aspirante catena discendente infinita: il termine senza argomenti c.

Parleremo più in dettagio delle segnature quando tratteremo l'induzione strutturale 2.2.3.

Esempio 2.10 (Ordinamento Lessicografico)
Vediamo un ultimo esempio di relazione ben fondata: l'insieme è quello delle coppie di interi (per semplicità), e la relazione è l'ordinamento lessicografico di questi interi. è cioè un ordinamento tale che

- $\forall m, m'. \langle n, m \rangle \prec \langle n+1, m' \rangle$
- $\langle n, m \rangle \prec \langle n, m+1 \rangle$

Anche in questo caso la relazione è ben fondata: qualunque catena discendente non può andare sotto la coppia $\langle 0, 0 \rangle$ e, non essendoci ovviamente cicli, questo basta a dimostrarlo.

Esempio 2.11 (Numeri interi)
Il minore stretto aritmetico sull'insieme dei numeri interi $\mathbb{Z}$ non è *una relazione ben fondata: esiste infatti una catena discendente infinita visto che, contrariamente a quanto accade con i numeri naturali, non esiste un limite inferiore all'insieme* $\mathbb{Z}$ *(ovvero, per ogni elemento della successione ne esiste sempre uno minore).*

Vediamo un infine teorema che lega una relazione con la propria chiusura transitiva.

Teorema 2.12 (Fondatezza della chiusura transitiva di $\prec$)
Una relazione $\prec$ è ben fondata se e solo se la sua chiusura transitiva $\prec^+$ è ben fondata.

Dimostrazione. La dimostrazione dritta è ovvia, in quanto se $\prec$ è ben fondata è ovvio che anche la sua chiusura transitiva, visto che ne è solo una scrittura compatta.
Dimostriamo il senso contrario della doppia implicazione prendendo una catena discendente infinita con $\prec^+$

$$a_0 \prec^+ a_1 \prec^+ a_2 \cdots$$

Questa può essere ovviamente trasformata in una catena discendente infinita con $\prec$ in questa maniera

$$a_0 = a_{00} \prec a_{01} \prec \cdots \prec a_{0n} = a_1 = a_{10} \prec a_{11} \prec \ldots \prec a_{1n} = a_2 = a_{21} \prec \cdots$$

2.1.2. L'induzione ben fondata di Noether

Passiamo a definire il principio di induzione ben fondata di Noether, che è la base di tutti i metodi di induzione che vedremo nella prossima sezione.

Teorema 2.13
Data una relazione ben fondata $\prec$ su un insieme A

$$(\forall a \in A.\ (\forall b \prec a.\ P(b)) \rightarrow P(a)) \Leftrightarrow (\forall a \in A.P(a))$$

Dimostrazione. Dimostriamo entrambe le implicazioni

$\Leftarrow$ ovvio: se $\forall a.P(a)$ allora la parte destra dell'implicazione da dimostrare è sempre vera, quindi ovviamente sarà vera tutta l'implicazione

$\Rightarrow$ prendiamo l'insieme $Q = \{a \in A | \neg P(a)\}$ di tutti gli elementi di A per i quali non vale P(a). Dato che $\prec$ è ben fondata, per il lemma 2.7 questo sottoinsieme avrà un elemento minimale m per il quale vale $\neg P(m)$: per costruzione di Q, ho che $\forall b \prec m.P(b)$ in quanto m è il minimo elemento per il quale P non vale. Ma questo contraddice l'ipotesi, visto che dovrebbe valere $\forall a \in A.(\forall b \prec aP(b)) \rightarrow P(a)$. Dobbiamo concludere che $\forall a \in A.P(a)$.

2.2. Altri metodi di induzione

Vediamo adesso alcuni altri metodi di induzione che sono in realtà versioni meno generali del principio di induzione di Noether: le utilizzeremo per dimostrare alcune proprietà successivamente.

2.2.1. Induzione matematica debole

L'induzione matematica debole è un principio che si applica per tutte le proprietà sull'insieme $\mathbb{N}$ dei numeri naturali.

Vedendola come caso speciale dell'induzione ben fondata, basta prendere come insieme A e come relazione $\prec$

$$A = \omega \qquad n \prec m \leftrightarrow m = n + 1$$

Vediamo l'enunciato.

Teorema 2.14 (Induzione matematica debole)
$(P(0) \wedge P(n) \rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}.P(n)$

2.2.2. Induzione matematica forte

L'induzione matematica forte è un'estensione dell'induzione matematica debole ed è ottenuta dal principio di induzione di Noether prendendo la relazione

$$n \prec m \leftrightarrow m = n + k$$

sull'insieme $\mathbb{N}$. Vediamo l'enunciato.

Teorema 2.15 (Induzione matematica forte)
$(P(0) \wedge (\forall i \leq n.P(n)) \rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}.P(n)$

2.2.3. Induzione strutturale

L'induzione strutturale è un'altra specializzazione dell'induzione ben fondata di Noether che riguarda l'insieme dei termini di una segnatura e la relazione termine-sottotermine che li lega.

Definizione 2.16 (Termini su una segnatura tipata)
Per definire i termini su di una segnatura tipata abbiamo bisogno di

- *un insieme di* sort S *che contiene i tipi che potranno assumere i nostri termini*
- *una segnatura* $\Sigma = \{\Sigma_{s_1 \dots s_n, s}\}$ *con* $s_i, s \in S$ *che è un insieme di operatori ognuno dei quali prende in ingresso* n *parametri* s_i *e ritorna come risultato un* s*, tutti naturalmente appartenenti ai tipi definiti in* S

Ora possiamo definire l'insieme dei termini della segnatura come

$$T_\Sigma = \{T_{\Sigma,s}\} \quad s \in S$$

e sono cioè istanze degli operatori definiti in Σ*. Vediamo come costruire i termini*

$$\frac{t_i \in T_{\Sigma,s_i} \quad i = 1, \dots, n \quad f \in \Sigma_{s_1,\dots,s_n,s}}{f(t_1, \dots, t_n) \in T_{\Sigma,s}}$$

A questo punto prendendo come insieme l'insieme dei termini appena definito e come relazione ben fondata la relazione termine-sottotermine

$$t_i < f(t_1, \dots, t_n) \quad i = 1, \dots, n$$

possiamo definire l'induzione strutturale.

Definizione 2.17 (Induzione strutturale)

$$\frac{\forall t \in T_{\Sigma} \quad (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_{\Sigma}.\ P(t)}$$

o in versione più estesa

$$\frac{\forall s_i \in S \quad i = 1, \ldots, n. \quad \forall s \in S. \quad \forall f \in \Sigma_{s_1 \ldots s_n, s}. \quad \forall t_i \in T_{\Sigma, s_i} \quad i = 1, \ldots, n. \quad P(t_1) \wedge \ldots \wedge P(t_n) \Rightarrow P(f(t_1, \ldots, t_n))}{\forall t \in T_{\Sigma}.\ P(t)}$$

Esempio 2.18 (Segnatura in IMP)
Per IMP abbiamo $S = \{Aexp, Bexp, Com\}$ *e l'operazione ";" è del tipo* $-;- \in \Sigma_{ComCom,Com}$*, o anche* $- = - \in \Sigma_{AexpAexp,Bexp}$.
Per costruire $T_{\Sigma,s}$ *è necessario partire da qualcosa senza argomenti, ad esempio:*

$$skip \in Com \qquad \frac{skip \in Com \quad x := a \in Com}{skip; x := a \in Com}$$

Per ricollegarci all'induzione matematica, potremmo immaginare di avere due sort $\Sigma_0 = \{0\}$ e $\Sigma_1 = \{succ\}$ e a questo punto la dimostrazione da effettuare sarebbe la seguente:

$$\frac{P(0) \quad P(n) \Rightarrow P(succ(n))}{\forall k \in \omega.\ P(k)}$$

L'induzione strutturale si applica ad esempio alle grammatiche: prendiamo ad esempio la produzione relativa alle espressioni aritmetiche in IMP

$$a :: n|x|a_0 + a_1|a_0 - a_1|a_0 \times a_1$$

Supponiamo di voler dimostrare una proprietà di a. Come possiamo fare? Semplice, basta dimostrare che questa proprietà vale *per tutte le sue produzioni*. In altre parole, se vogliamo dimostrare P(a) dobbiamo saper dimostrare che valgono

- P(n) per ogni intero n
- P(x) per ogni identificatore x
- P($a_0 + a_1$) assumendo che valgano P(a_0) e P(a_1)
- P($a_0 - a_1$) assumendo che valgano P(a_0) e P(a_1)
- P($a_0 \times a_1$) assumendo che valgano P(a_0) e P(a_1)

Esempio 2.19 (Induzione strutturale)
Prendiamo il caso delle espressioni aritmetiche visto sopra, e dimostriamo che

$$\forall a \in Aexpr, \sigma \in \Sigma, m \in \mathbb{N}, m' \in \mathbb{N}. \langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m' \Rightarrow m = m'$$

In altre parole, presa un'espressione aritmetica ed una memoria σ *essa restituirà sempre uno ed un solo valore. Dimostriamolo con induzione strutturale.*

$a \equiv n$ per sviluppare un numero intero esiste una ed una sola regola, che restituisce sempre lo stesso valore. Quindi ovviamente m = m'

$a \equiv x$ come sopra, esiste un unica regola per sviluppare questa forma di a. Questa regola restituisce $\sigma(x)$, ma visto che σ è uguale in entrambi i casi ovviamente otterremo lo stesso valore

$a \equiv a_0 + a_1 | a_0 - a_1 | a_0 \times a_1$ per definizione otterremo due risultati $m = m_0 + m_1$ e $m' = m'_0 + m'_1$. Per ipotesi dell'induzione strutturale però $m_0 = m'_0$ e $m_1 = m'_1$, quindi ovviamente anche $m = m'$ qualunque sia l'operazione aritmetica applicatavi.

2.2.4. Induzione sulle derivazioni

Considerare solamente l'induzione ben fondata non è talvolta sufficiente: a volte serve un modello di ragionamento che si basa sulle regole di inferenza stesse. Abbiamo visto cosa siano una regola di inferenza ed una derivazione nelle definizioni 1.2 e 1.3.

Proviamo a definire una relazione ben fondata sull'insieme delle derivazioni.

Definizione 2.20 (Sottoderivazione immediata)
Si dice che d' è sottoderivazione immediata *di d, in simboli $d' \prec_1 d$, se e solo se d è della forma (D/y) e $d' \in D$.*

Esempio 2.21 (Sottoderivazione immediata)
Nella derivazione

$$\cfrac{\cfrac{}{\langle 1, \sigma \rangle \to 1}\, num \qquad \cfrac{}{\langle 2, \sigma \rangle \to 2}\, num \qquad 1 + 2 = 3}{\langle 1 + 2, \sigma \rangle \to 3}\, sum$$

le regole num sono sottoderivazioni immediate della regola sum.

Definiamo adesso il concetto di sottoderivazione propria.

Esempio 2.22 (Sottoderivazione propria)
Si dice che d' è sottoderivazione propria si d se e solo se $d' \prec d$, dove $\prec = \prec_1^+$

Visto che esistono degli elementi minimi di $\prec$, cioè gli assiomi, entrambe le relazioni sono ben fondate e possono quindi essere utilizzate per dimostrazioni per induzione. Vediamo un esempio.

Esempio 2.23 (Dimostrazione mediante induzione sulle derivazioni)
Vogliamo dimostrare che la derivazione di un teorema in IMP è deterministica. In simboli, presi un comando c ed una memoria σ, vogliamo dimostrare la seguente proprietà

$$P(d) = \forall \sigma, \sigma', \sigma'' \in \Sigma . d \vdash_R \langle c, \sigma \rangle \to \sigma' \wedge d \vdash_R \langle c, \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$$

Lo dimostreremo per induzione sulle derivazioni*: dobbiamo quindi dimostrare che $(\forall d' \prec d . P(d')) \to P(d)$. E questo va fatto per tutte le possibili forme di d. Vediamone alcune.*

CHECKME: Controllare se è una boiata.
Ricordiamo che dire che $(\forall d' \prec d.P(d')) \rightarrow P(d)$, ci permette dire che per tutti gli elementi che precedevano d, valeva la proprietà del determinismo. Possiamo affermare quindi che l'induzione sulle derivazioni non è altro che un induzione ben fondata in cui l'operatore $\prec$ è definito come sopra. I puntini sopra le derivazioni stanno ad indicare proprio il fatto che non sappiamo a priori come è fatta la derivazione.

- *Se d è un comando* **skip**, *abbiamo che $\sigma' = \sigma'' = \sigma$ in quanto lo skip non modifica lo stato.*
- *Se d è un comando di assegnamento, abbiamo due regole della forma*

$$\frac{\dfrac{\vdots \quad \vdots}{\langle a, \sigma\rangle \rightarrow m}}{\langle X := a, \sigma\rangle \rightarrow \sigma[^m/_X]} \qquad \frac{\dfrac{\vdots \quad \vdots}{\langle a, \sigma\rangle \rightarrow m'}}{\langle X := a, \sigma\rangle \rightarrow \sigma[^{m'}/_X]}$$

  Ma in σ il valore dell'espressione aritmetica a sarà lo stesso in entrambe le espressioni, dato che la loro valutazione è deterministica, e di conseguenza sarà uguale anche la memoria restituita dal comando!

- *Se d è una concatenazione di comandi, abbiamo due regole della forma*

$$\frac{\dfrac{\vdots}{\langle c_1, \sigma\rangle \rightarrow \sigma_1} \quad \dfrac{\vdots}{\langle c_2, \sigma_1\rangle \rightarrow \sigma'}}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma'} \qquad \frac{\dfrac{\vdots}{\langle c_1, \sigma\rangle \rightarrow \sigma_2} \quad \dfrac{\vdots}{\langle c_2, \sigma_2\rangle \rightarrow \sigma''}}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma''}$$

  Ovviamente avremo che
  - *le derivazioni con le quali si deduce $\langle c_1, \sigma\rangle \rightarrow \sigma_1$ e $\langle c_2, \sigma_1\rangle \rightarrow \sigma'$ sono sottoderivazioni proprie della derivazione che stiamo analizzando, quindi per loro vale l'ipotesi induttiva*
  - *per questo possiamo concludere che $\sigma_1 = \sigma_2$, in quanto entrambe lavorano sulla stessa coppia comando/memoria*
  - *ma allora anche $\sigma' = \sigma''$, come volevamo dimostrare!*

- *Se d è un comando condizionale, dobbiamo considerare i due casi in cui la guardia booleana sia vera o falsa. Supponiamo che sia vera: abbiamo ancora due regole della forma*

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \rightarrow \mathbf{true}} \quad \dfrac{\vdots}{\langle c_1, \sigma\rangle \rightarrow \sigma'}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \rightarrow \sigma'} \qquad \frac{\dfrac{\vdots}{\langle b, \sigma\rangle \rightarrow \mathbf{true}} \quad \dfrac{\vdots}{\langle c_1, \sigma\rangle \rightarrow \sigma''}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \rightarrow \sigma''}$$

  Siamo ancora nella situazione del punto precedente: la regola per derivare $\langle c_1, \sigma\rangle \rightarrow \sigma'$ è sottoderivazione propria della regola in esame, quindi possiamo applicare l'ipotesi induttiva e concludere che $\sigma' = \sigma''$

- *Infine, se d è un comando di tipo while dobbiamo ancora considerare i due casi in cui la guardia sia vera o falsa.*

  Il caso in cui la guardia è falsa è banale, in quanto lo stato non viene modificato e, come nel caso dello skip, risulta $\sigma' = \sigma'' = \sigma$. Interessante è invece il caso nel quale la guardia è vera: abbiamo le due regole

$$\frac{\langle b, \sigma\rangle \rightarrow \mathbf{true} \quad \langle c, \sigma\rangle \rightarrow \sigma_1 \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_1\rangle \rightarrow \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{true} \quad \langle c,\sigma\rangle \to \sigma_2 \quad \langle \textbf{while}\, b\, \textbf{do}\, c,\sigma_2\rangle \to \sigma''}{\langle \textbf{while}\, b\, \textbf{do}\, c,\sigma\rangle \to \sigma''}$$

Come sempre, di sopra abbiamo lo stesso while che troviamo di sotto. Tuttavia, per quanto possa sembrare strambo, il while di sopra è comunque sottoderivazione propria di quello di sotto (la definizione parla chiaro!), e quindi possiamo applicarvi l'ipotesi induttiva che, assieme all'applicazione su c, permette di affermare che anche il ramo true del while è deterministico, ponendo fine a questa faticosa dimostrazione.

2.2.5. Induzione sulle regole

Questo ultimo caso di induzione si applica ad insiemi di elementi definiti mediante regole. Si basa sull'idea che, immaginando di avere un insieme di regole di transizione che definiscono quali elementi appartengano ad un certo insieme, si debba dimostrare che l'applicazione di una regola non pregiudichi la veridicità di un predicato che vogliamo dimostrare. Formalizziamo qualche concetto.

Una regola è un oggetto della forma $(\emptyset/y)$ oppure $(\{x_1,\ldots,x_n\}/y)$. Dato questo insieme R di regole, definiamo *l'insieme dei teoremi* come

$$I_R = \{x \mid \vdash_R x\}$$

cioè l'insieme degli elementi per i quali esiste una derivazione.

L'induzione sulle regole punta a dimostrare che tutti gli elementi di I_R godono di una certa proprietà P, e lo fa dimostrando che

- per ogni assioma $(\emptyset/y)$ vale P(y)
- per ogni regola $(\{x_1,\ldots,x_n\}/y)$ vale $\forall 1 \le i \le n.(x_i \in I_R \wedge P(x_i)) \Rightarrow \forall y \in I_R.P(y)$

Caso speciale in cui $P(\{d_1 \cdots, d_n/y\})$ si riduce a $P(y)$
Non si considerano tutte le premesse, ma solo quelle derivabili

$$\frac{\forall (X/y) \in R \quad X \subseteq I_R \quad \cdot (\forall x \in X \cdot P(x)) \Longrightarrow P(y)}{\forall x \in I_R.P(x)}$$

L'induzione sulle regole vale $d \vdash y\ p(d) \Leftrightarrow\ Q(y)\ d_1, d_n$: sono dimostrazioni. x_1, x_n: premesse r regola di inferenza

Delle regole che non sono raggiungibili, non serve dimostrarle in una dimostrazione.
Principio di induzione sulle regole:
Qui le formule non c'entrano niente (Viene chiesto all'esame), non si fa induzione sulle formule, c'entrano le derivazioni che sono fatte con le regole. $x_i \in I_r$ si può usare o no, non è obbligatorio.

2.3. Ricorsione ben fondata (Well founded recursion)

Passiamo al concetto di ricorsione ben fondata. Una ricorsione si dice *ben fondata* quando riguarda una funzione che nella propria definizione chiama ricorsivamente se stessa su valori minori secondo una certa relazione ben fondata. L'insieme delle funzioni che seguono il principio della ricorsione ben fondata è detto insieme delle funzioni *primitive ricorsive*.

Esempio 2.24 (Ricorsione ben fondata)
Prendiamo la formula di Peano per il calcolo del prodotto

$$P(0,y) = 0$$
$$P(x+1,y) = y + P(x,y)$$

Questa formula segue il principio dell'induzione ben fondata, in quanto è definita sulla base di se stessa applicata su valori minori secondo una relazione ben fondata (per la precisione l'ordine lessicografico).

Passiamo ad una formalizzazione di questo concetto. Definito il seguente insieme

Definizione 2.25 (Insieme dei predecessori)
L'insieme dei predecessori di un insieme I è l'insieme
$\prec^{-1} I = \{b \in B | \exists b' \in I.\ b \prec b'\}$

Teorema 2.26 (Ricorsione ben fondata)
Sia $(B, \prec)$ una relazione ben fondata su B. Prendiamo una funzione $F(b, h)$ che restituisce valori in un insieme C, dove

- $b \in B$
- $h : \prec^{-1} \{b\} \to C$

Allora esiste un'unica funzione $f : B \to C$ tale che

$$\forall b \in B.\ f(b) = F(b, f \upharpoonright \prec^{-1} \{b\})$$

Questo significa se definiamo la funzione ricorsiva f per un certo valore b solo in funzione dei suoi predecessori, allora f può essere espressa tramite un'*unica* funzione.

Nota: la notazione $f \upharpoonright B'$ indica che la funzione $f : B \to C$ viene ristretta ai soli valori del dominio appartenenti a B'.

In seguito denoteremo la semantica delle funzioni ricorsive tramite la teoria del punto fisso, tuttavia la ricorsione ben fondata risulta un metodo più semplice ed altrettanto valido in alcuni casi.

Vediamo qualche esempio per chiarire bene il concetto.

Esempio 2.27 (Formalizzazione della ricorsione ben fondata)
Applichiamo la definizione formale alla formula di Peano vista nell'esempio 2.24. Per comodità di notazione, essendo y di fatto un parametro della funzione e non una variabile, indicheremo $p(x, y)$ come $p_y(x)$.

Abbiamo

- $P_y(0) = F_y(0, P_y \upharpoonright \emptyset) = 0$
- $P_y(x + 1) = F_y(x + 1, P_y \upharpoonright \prec^{-1} \{x + 1\}) = y + P_y(x)$

Notare che stiamo definendo *la F nei due differenti casi, e che questa F vale per* qualunque *x gli venga passato.*

Esempio 2.28 (Ricorsione ben fondata per una visita di albero in Lisp)
Lisp

$$sum(x) = \textbf{if}\ atom(x) Henv(x)\ \textbf{else}\ sum(car(x)) + sum(cdr(x))$$

Somme di un generico albero
car (x) = sottoalbero sinistro di x
cdr (x) = sottoalbero destro di x
cons(x,y) = costruttore

$$\begin{array}{|l} sum(i) = i \\ sum(cons(x, y)) = sum(x) + sum(y) \end{array}$$

Segnatura: $B = T_\sigma$ *(alberi binari)*
Algebra
$c = w$
$\Sigma_0 = \{0, 1, ...\}$
$\Sigma_2 = cons$
$x_i \prec cons(x_1, x_2)$ *(relazione ben fondata)*

$$\begin{array}{|l} F(i, \phi) = i \\ F(cons(x, y), sum) = sum(x) + sum(y) = sum(cons(x, y)) \end{array}$$

A questo punto possiamo applicare il teorema: calcola uno un solo risultato e non si blocca mai.
La semantica denotazionale è definita per ricorsione strutturale.

Esempio 2.29 (Funzione di Ackermann)

$$\begin{array}{|l} ack(0, 0, z) = z \\ ack(0, y + 1, z) = f(0, y, z) + 1 \\ ack(1, 0, z) = 0 \\ ack(x + 2, 0, z) = 1 \\ ack(x + 1, y + 1, z) = ack(x, ack(x + 1, y, z), z) \end{array}$$

ack(0, x, y) = x + y
ack(1,x+1,y) = ack(0, ack(1, x, y), y) = ack(1,x,y) +y => ack(1, x, y) = x y
ack(2, 0, y) 1 ack(2, x+1, y) = ack(2, x, y) y (Esponenziale)
$ack(2, x, y) = y^x$
In questo caso non possiamo trattare gli argomenti come parametri, ma sono variabili e quindi cambiamo l'ordinamento dai naturali a quello lessicografico.

3. Ordinamenti parziali e teorema del punto fisso

3.1. Basi degli ordinamenti parziali

Intuizione:
Gli ordini parziali completi corrispondono a tipi di strutture dati che possono essere usati come ingresso o uscita di un calcolo. Le funzioni calcolabili sono modellata come funzioni continue tra questi dati. Gli elementi di un cpo vengono considerati come punti di informazione e l'ordinamento $x \sqsubseteq y$ significa che x approssima y (o x ha la stessa o meno informazione di y), $\bot$ è quindi il punto di minore informazione.

3.1.1. Ordinamenti parziali e totali

Finora abbiamo visto solamente come studiare definizioni date mediante ricorsione ben fondata, come ad esempio il mondo delle espressioni aritmetiche o quello delle booleane. Queste definizioni sono estremamente comode, ma hanno un potere espressivo limitato all'insieme di funzioni cosiddette primitive ricorsive.

Senza addentrarci troppo nei dettagli, le funzioni primitive ricorsive sono tutte quelle esprimibili mediante programmi del nostro linguaggi IMP nel quale al posto del comando while viene utilizzato un comando FOR che itera su collezioni *finite* di elementi[1]. Un linguaggio cosí modificato non puó andare in ciclo, o in altre parole, se definiamo una relazione $\prec$ sull'insieme delle regole del nostro linguaggio tale che una formula ben fondata è maggiore nell'ordinamento rispetto ad una sua sottoderivazione propria, non vengono generate catene discendenti infinite.

Tuttavia, questo non ci basta. Le funzioni primitive ricorsive sono tante, ma non sono sufficienti a descrivere l'intero spettro delle funzioni totali. Per esprimere alcune di queste funzioni (ad esempio, la funzione di Ackermann che abbiamo visto nello scorso capitolo) avremo bisogno di costrutti come il **while** che possono non terminare, e che quindi non rientrano nelle nostre funzioni primitive ricorsive. è per questo che abbiamo bisogno di una teoria differente, che consideri ordinamenti e catene differenti da quelle che abbiamo visto finora. è la teoria del punto fisso, che definiremo meglio a metà capitolo.

La strada per arrivare al punto fisso è lunga, dunque cominciamo subito con una serie di definizioni che ci serviranno strada facendo.

Definizione 3.1 (Ordinamento Parziale)
Un ordinamento parziale $(P, \sqsubseteq)$ *è una relazione* $\sqsubseteq \subseteq P \times P$ *tale che valgono*

- *Proprietà riflessiva:* $p \sqsubseteq p$
- *Proprietà antisimmetrica:* $p \sqsubseteq q \quad q \sqsubseteq p \Longrightarrow p = q$
- *Proprietà transitiva:* $p \sqsubseteq q \quad q \sqsubseteq r \Longrightarrow p \sqsubseteq r$

L'insieme P è detto poset *(partially ordered set).*

Notiamo che un qualunque sottoinsieme di un insieme parzialmente ordinato ne eredita la struttura ed è quindi ancora un insieme parzialmente ordinato.

[1] In particolare, non parliamo del for del linguaggio Java^{TM}o del C che sono di fatto una riscrittura del while e possono andare in loop, bensí del comando **foreach** utilizzato in linguaggi come C# che itera su collezioni finite di elementi: un comando del genere, intuitivamente, non può andare in loop a meno che non sia il corpo ad andarci!

Esempio 3.2 (Ordinamento parziale)
L'ordinamento $\langle \mathbb{N}, \leq \rangle$ è un ordinamento parziale: infatti

- *vale la proprietà riflessiva, in quanto $\forall n \in \mathbb{N}.n \leq n$*
- *vale la proprietà antisimmetrica, in quanto è ovvio che $\forall n, m \in \mathbb{N}.n \leq m \vee m \leq n \vee m = n$ per la proprietà della relazione $\leq$*
- *allo stesso modo vale ovviamente la proprietà transitiva*

Esempio 3.3 (Ordinamento non parziale)
L'ordinamento $\langle \mathbb{N}, < \rangle$ non è un ordinamento parziale: infatti

- *vale la proprietà transitiva, per le proprietà di $<$*
- ***non** vale la proprietà antisimmetrica, in quanto $p < q \quad q < p \not\Longrightarrow p = q$ (in particolare, p e q in questo caso non esistono!)*
- ***non** vale neppure la proprietà riflessiva: un numero non è strettamente minore di se stesso*

Vale solo una proprietà su tre, quindi l'ordinamento ovviamente non è parziale.

Definizione 3.4 (Ordinamento totale)
Un ordinamento su un insieme P è detto totale se è un ordinamento parziale (e dunque valgono le proprietà riflessiva, antisimmetrica e transitiva) ed inoltre vale

$$\forall p_1, p_2 \in P.x = y \vee x \sqsubseteq y \vee y \sqsubseteq x$$

Un insieme sul quale è definito un ordinamento totale è detto insieme totalmente ordinato.
Vediamo qualche ordinamento particolare che utilizzeremo in futuro.

Esempio 3.5 (Ordinamento discreto)
Un ordinamento discreto è un ordinamento in cui la relazione di ordinamento è definita come $\forall x.x \sqsubseteq x$. Questo ordinamento é ovviamente parziale: la proprietà riflessiva vale sicuramente, così come la proprietà transitiva (in quanto non é definito ordinamento per elementi diversi fra di loro). Infine la proprietà antisimmetrica vale, in quanto l'unico caso in cui due elementi sono messi a confronto é quello in cui i due elementi sono uguali:

Esempio 3.6 (Ordinamento piatto)
Un ordinamento piatto è un ordinamento in cui esiste un elemento $\bot$ tale che $\forall x.\bot \sqsubseteq x \wedge x \sqsubseteq x$ e non c'è nessun altra relazione. In questo caso la proprietà transitiva non ha senso, così come quella antisimmetrica (in quanto non possono esistere due elementi reciprocamente uno minore dell'altro, per definizione). La proprietà riflessiva, infine, viene dalla definizione.

Per ricollegarci alle relazione ben fondate, un ordine parziale è detto ben fondato se il corrispondente ordine stretto è una relazione ben fondata.

Enunciamo adesso un teorema che ci risulterá comodo in seguito.

Teorema 3.7 (Sottoinsiemi di un insieme totalmente ordinato)
Se A è un insieme totalmente ordinato, $\forall B \subseteq A.B$ è un insieme totalmente ordinato rispetto alla stessa relazione.

Utilizzeremo questo teorema quando parleremo di catene: essendo esse sottoinsiemi completamente ordinati di ordinamenti parziali, sapremo che

- ogni sottoinsieme di un ordinamento totale è una catena
- ogni sottoinsieme di una catena è una catena

3.1.2. Diagrammi di Hasse

Rappresentare un ordinamento è una cosa piuttosto scomoda. Immaginiamo di voler rappresentare la relazione con un grafo, ad esempio sui numeri naturali: dall'n-esimo elemento dovrebbero partire n-1 frecce verso gli elementi minori rispetto a lui! Ci serve una notazione più compatta. Vediamone una.

Definizione 3.8 (Diagramma di Hasse)
Definiamo una relazione R tale che:

$$\frac{x \sqsubseteq y \quad y \sqsubseteq z \quad y \neq z \neq x}{xRz} \qquad xRx$$

Una relazione $\sqsubseteq$ da cui si eliminano gli archi dovuti alla transitività e alla riflessività è un diagramma di Hasse

$$H = \sqsubseteq - R$$

In parole povere, togliamo dal grafo completo tutte le frecce riflessive e tutte le frecce che collegano due nodi fra i quali esiste un cammino più lungo.

Come facciamo ad essere sicuri che questa notazione descrive completamente la nostra relazione di partenza? Vale il seguente teorema.

Teorema 3.9 (Equivalenza fra relazione e chiusura di Hasse)
Se P è un insieme finito, la chiusura transitiva e riflessiva di un diagramma di Hasse è uguale alla relazione di ordine parziale originaria.

$$H^* = \sqsubseteq$$

Questo discorso, tuttavia, non vale se P è un insieme infinito: prendiamo ad esempio la relazione $(\omega, \leq)$, ed aggiungiamoci ∞. Infine estendiamo la relazione $\leq$ con: $n \leq \infty$ e $\infty \leq \infty$.

Applicando Hasse a questo insieme eliminiamo tutti gli archi tra ∞ e n, ma quando richiudiamo con * ritroviamo solo gli archi che collegano i vari n tra di loro mentre ∞ rimane staccato.

Ricordiamo brevemente il concetto di chiusura di una relazione

Definizione 3.10 (Chiusura)
Una chiusura di una relazione R è una relazione R^* che rispetta le seguenti proprietà:

- xR^*x
- $\dfrac{xR^*y \quad yRz}{xR^*z}$

Vediamo ora alcuni punti di interesse dei nostri ordini parziali.

Definizione 3.11 (Minimo di un insieme)
$m \in S$ è minimo in $(S, \sqsubseteq)$ se $\forall s \in S.\ m \sqsubseteq s$

Notare che il minimo deve essere in relazione con *tutti* gli elementi dell'insieme.

Teorema 3.12 (Unicità del minimo)
Il minimo, se esiste, è unico.

Dimostrazione. Banalmente $m_1 \sqsubseteq m_2 \quad m_2 \sqsubseteq m_1$ per la proprietà antisimmetrica implica $m_1 = m_2$.

Duale è la definizione di un elemento *massimo*.

Definizione 3.13 (Elemento Minimale di un insieme)
$m \in S$ si dice elemento minimale di $(S, \sqsubseteq)$ se $\forall s \in S.\ s \sqsubseteq m \Rightarrow s = m$

Duale è la definizione di un elemento *massimale*.
In generale elementi minimo e minimale (massimo e massimale) non sono la stessa cosa, infatti al contrario del minimo il minimale non deve essere in relazione con tutti gli altri elementi e quindi possono esistere più elementi minimali non in relazione tra loro.
Naturalmente nel caso di un ordinamento totale le due definizioni coincidono.

Vediamo ora dei concetti simili relativi ai sottoinsiemi di un ordinamento parziale.

Definizione 3.14 (Maggiorante (upper bound))
Dato un ordinamento parziale $(P, \sqsubseteq)$ ed un sottoinsieme $X \subseteq P$ diciamo che $p \in P$ è upper bound *di X se e solo se*

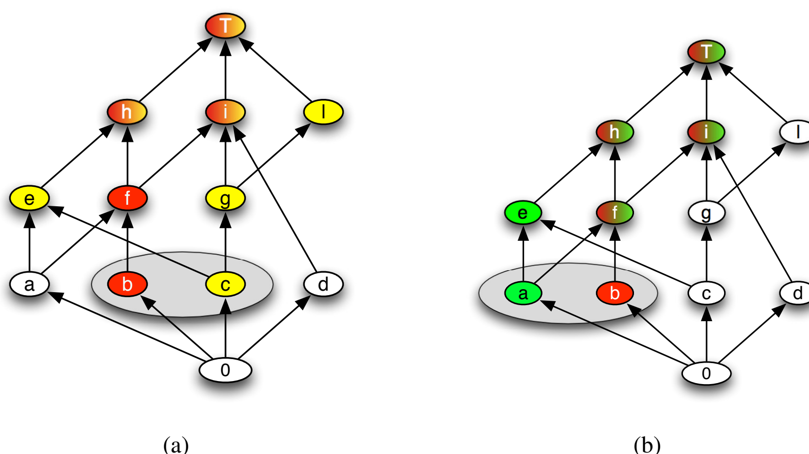
$$\forall q \in X. q \sqsubseteq p$$

É da notare una cosa importante: *non è detto* che l'upper bound appartenga ad X!
Fra tutti gli upper bounds di un sottoinsieme ne può esistere uno che ci interesserà particolarmente.

Definizione 3.15 (Minimo dei maggioranti (LUB))
Dato un ordinamento parziale $(P, \sqsubseteq)$ ed un sottoinsieme $X \subseteq P$ il valore $p \in P$ è least upper bound *di X, e si indica con $lub(X)$, se e solo se*

- *p è un upper bound di X*
- *per tutti gli upper bounds $q \neq p$ di X, $p \sqsubseteq q$*

Vediamo due esempi per chiarire cosa sia il lub e perché potrebbe non esistere.
Nella figura 3.1 abbiamo due situazioni differenti di sottoinsiemi di un insieme sul quale è stato definito un ordinamento parziale (del quale mostriamo il diagramma di Hasse).
Prendendo il sottoinsieme evidenziato dall'ovale grigio nella figura di sinistra, composto dagli elementi b e c, abbiamo che l'insieme degli upper bounds è l'insieme formato dagli elementi i, h e $\top$, in quanto unici



(a) (b)

Figura 3.1.: Due sottoinsiemi, dei quali il primo ha LUB e il secondo no.

elementi ad essere maggiori di *entrambi gli elementi dell'insieme*. Questo insieme non ha un minimo: h ed i sono minori rispetto a $\top$, ma non sono in relazione fra di loro.

Prendendo invece il sottoinsieme evidenziato dall'ovale grigio in figura di destra, composto dagli elementi a e b, abbiamo che l'insieme degli upper bounds comprende anche l'elemento f che e' minore rispetto a tutti gli altri elementi degli insieme dei maggiori. Possiamo quindi dire che f è lub dell'insieme.

Dualmente possiamo definire i concetti di *Minorante* e *Massimo dei minoranti*.

3.1.3. Catene

Passiamo a definire un altro concetto importante, cioè quello di *catena*.

Definizione 3.16 (Catena)
Dato un insieme parzialmente ordinato $(P, \sqsubseteq)$, un sottoinsieme D di P è una catena se

$$\forall i_1, i_2 \in D \quad i_1 \sqsubseteq i_2 \vee i_2 \sqsubseteq i_1$$

ovvero una catena è un sottoinsieme totalmente ordinato *di P.*

Una catena $(D, \sqsubseteq)$ può anche essere una sequenza infinita di elementi $d \in D$

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

ovvero

$$\{d_i\}_{i \in \omega}.d_i \sqsubseteq d_{i+1}$$

Possiamo vedere queste catene infinite come funzioni $\sigma : \omega \to D$ che associano a infiniti passi gli elementi di D. Vediamo qualche proprietà di tali catene

Teorema 3.17 (Catene di un insieme finito)
Se D è un insieme finito con ordinamento parziale allora ogni sua catena è finita*.*

Vedremo in seguito che per questa ragione D è un ordine parziale completo.

Nel caso in cui una catena sia finita, si dice che si *stabilizza* su un particolare valore. Si crea così una catena del tipo

$$d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d \sqsubseteq d \sqsubseteq \ldots$$

nella quale un certo valore d si ripete all'infinito a partire da un certo momento in poi.
Notare che anche se avessimo dei cicli del tipo

$$d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d \sqsubseteq d_h \sqsubseteq d_{h+1} \sqsubseteq \ldots \sqsubseteq d$$

per la proprietà transitiva e antisimmetrica deve valere per forza

$$d_h = d_{h+1} = \ldots = d$$

in quanto se una di quelle uguaglianze cadesse avremmo un elemento in D che sarebbe sia maggiore che minore di d, il che contraddice l'antisimmetria.

Definizione 3.18 (Limite di una catena)
Nel caso in cui la catena sia finita, l'elemento in cui si stabilizza viene dette limite della catena ed è proprio il suo LUB.

Definizione 3.19 (Altezza di un insieme)
Un insieme parzialmente ordinato $(P, \sqsubseteq)$ *ha* altezza *finita se e solo se tutte le catene di P sono finite.*

Vediamo un esempio.

Esempio 3.20 (Insieme infinito con catene finite)
L'insieme dei singoletti numeri naturali, con top e bottom.

$$(\omega \cup \{\bot, \top\}, \leq) \quad \forall n \in \omega : \bot \sqsubseteq n \sqsubseteq \top$$

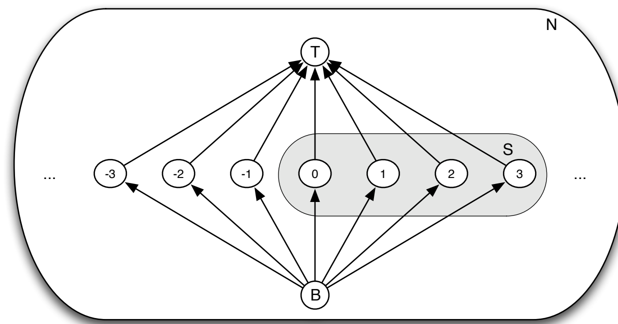
Passiamo ad un altra definizione.

Definizione 3.21 (Sottoinsieme diretto)
Sia $(P, \sqsubseteq)$ *un insieme parzialmente ordinato.*
Un sottoinsieme S di P si dice diretto se ogni sottoinsieme finito di S ha un least upper bound in S.

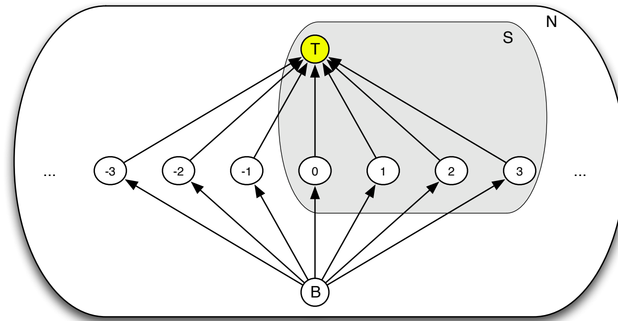
La figura 3.2 dovrebbe far capire la differenza fra un sottoinsieme diretto ed uno non diretto.
Nella figura sopra il sottoinsieme S di N non è diretto: se noi prendiamo un sottoinsieme qualunque, questo sicuramente non avrà LUB in quanto l'unico elemento maggiore nell'ordinamento rispetto a tutti gli elementi è $\top$, che non appartiene al sottoinsieme S.
Nella figura sotto, invece, il sottoinsieme S di N comprende $\top$: qualunque sottoinsieme si prenda in S, esso avrà sicuramente top come LUB, in quanto è l'unico upper bound in comune fra gli elementi di N.

Definizione 3.22 (Insieme inclusivo)
Sia $(D, \sqsubseteq)$ *un cpo. Un sottoinsieme P di D è* inclusivo *(o ammissibile) se e solo se per ogni catena* $\{d_i\}$ *in D*

$$(\forall n \quad d_n \in P) \Longrightarrow \bigsqcup_{n \in \omega} d_n \in P$$



(a)



(b)

Figura 3.2.: Due sottoinsiemi, uno dei quali diretto e l'altro no

Vedremo che la nozione di insieme inclusivo ci tornerà utile quando parleremo di CPO, cioè ordinamenti parziali *completi*. Vedremo che se abbiamo un CPO $(D, \sqsubseteq)$, potremo affermare con certezza che un PO definito su un suo sottoinsieme con lo stesso ordinamento $(P \subseteq D, \sqsubseteq)$ é a sua volta completo solo se D é inclusivo.

Inoltre utilizzeremo questa definizione nel paragrafo 4.3 nel quale parleremo di induzione computazionale. Vediamo un esempio adesso di insieme non inclusivo.

Esempio 3.23 (Fairness, insieme non inclusivo)

$$(D, \sqsubseteq) \qquad D = \{a, b\}^* \cup \{a, b\}^\infty \qquad \alpha \sqsubseteq \alpha\beta \quad \alpha_\infty\beta = \alpha$$

Siano queste stringhe la sequenza di scelte di un arbitro che scelga a o b.
Una scelta si dice fair *se non accade infinitamente che ci siano solo a o b (quindi le uniche stinghe unfair sono quelle infinite).*
L'insieme delle stringhe fair non è inclusivo infatti scegliendo ad esempio

$$a \sqsubseteq aa \sqsubseteq aaa \sqsubseteq \dots$$

ciascuna di queste stringhe finite è fair perché può sempre essere essere seguita da una b mentre il limite di questa catena è una stringa infinita di a.

NOTA: le definizioni di insieme diretto e inclusivo sembrano molto legate se non equivalenti, bisogna fare luce!

Passiamo ad un importante teorema.

Teorema 3.24 (Catene di insiemi parzialmente ordinati)
Sia $(P, \sqsubseteq)$ un insieme parzialmente ordinato. Ogni catena C non vuota di P è un insieme diretto.

Dimostrazione. É facile vederlo. Una catena è un insieme totalmente ordinato per definizione: se, dato un sottoinsieme di C, costruiamo l'insieme degli upper bounds di tale insieme questo sarà a sua volta totalmente ordinato per il teorema 3.7. Quindi avrà un elemento minimo, che sarà LUB del sottoinsieme di C. ☠

3.1.4. Ordinamenti parziali completi (CPO)

Arriviamo adesso ad un concetto importantissimo. Stiamo per definire una classe di ordinamenti completi che sono centro dell'intero corso: gli ordinamenti parziali completi.

Un ordinamento parziale completo è una generalizzazione del reticolo completo, ed é definito come segue.

Definizione 3.25 (Ordinamento parziale completo (CPO))
Un ordinamento parziale è detto completo *se ogni sua catena ha lub.*

Definiamo inoltre il seguente:

Definizione 3.26 (Ordinamento parziale completo con bottom)
Un ordinamento parziale completo $(D, \sqsubseteq)$ è con bottom ($CPO_\perp$) se ha un elemento $\perp$ tale che $\forall x \in D. \perp \sqsubseteq x$

Ovviamente un reticolo completo è anche un ordinamento parziale completo con bottom, in quanto se tutti i sottoinsiemi possiedono glb e lub allora a maggior ragione li possiederanno le catene.

In seguito, indicheremo con CPO un ordinamento parziale completo e con $CPO_\perp$ un CPO con bottom.

Si noti che qualunque ordinamento parziale che possiede solamente catene finite è un CPO, e dunque un ordinamento parziale su un insieme finito è sicuramente un CPO.

L'importanza dei CPO sta nel fatto che potremo usarli per ammettere l'esistenza di un limite di una catena in un insieme con un numero infinito di elementi, come vedremo in seguito.

Esempio 3.27 (Ordinamento parziale non completo (1))
Vediamo un esempio di ordinamento parziale non completo.

$$(\omega, \leq) \qquad n_0 \leq n_1 \leq n_2 \quad |\{n_i\}| \text{ infinita}$$

Le catene sono infinite, dunque siamo sicuri che non abbiano limite. Tuttavia, se introduciamo un elemento aggiuntivo ∞, l'ordinamento parziale diventa completo: ∞ è maggiore rispetto a tutti gli elementi di ω e dunque risulta essere sicuramente un lub perfetto per le catene infinite.

Esempio 3.28 (Ordinamento parziale completo (2))
Prendiamo l'ordinamento parziale $(P, \leq)$ e modifichiamolo, prendendo $(P', \leq')$ con

- $P' = P \cup \{\infty_1, \infty_2\}$
- $a \leq b \Rightarrow a \leq' b, a \leq' \infty_1, a \leq' \infty_2, \infty_1 \leq' \infty_1, \infty_2 \leq' \infty_2$

Questo ordinamento parziale non è completo, perché nonostante ci siano 2 maggioranti ∞_1 e ∞_2 questi non sono confrontabili fra di loro e quindi non è possibile stabilire quale sia il lub.

Esempio 3.29 (Powerset con inclusione)
*Dato un insieme S, prendiamo come dominio l'insieme dei suoi sottoinsiemi (*Powerset*) e come relazione di ordinamento l'inclusione* $(2^S, \subseteq)$.

Questo insieme rispetta queste proprietà:

- *riflessività* $S \subseteq S$
- *anti-simmetria* $\left.\begin{array}{l} S_1 \subseteq S_2 \\ S_2 \subseteq S_1 \end{array}\right\} \Longrightarrow S_1 = S_2$
- *transitività* $S_1 \subseteq S_2 \subseteq S_3 \Longrightarrow S_1 \subseteq S_3$

Possiamo concludere che è un CPO, infatti ogni catena ha limite:

$$lub(S_0 \subseteq S_1 \subseteq S_2..) = \{d \mid \exists k.\ d \in S_k\} = \bigcup_{i \in \omega} S_i$$

Notiamo che in realtà il powerset con inclusione possiede anche un elemento $\bot = \emptyset$ *e come vedremo in seguito è anche un reticolo, un oggetto molto più strutturato del CPO.*

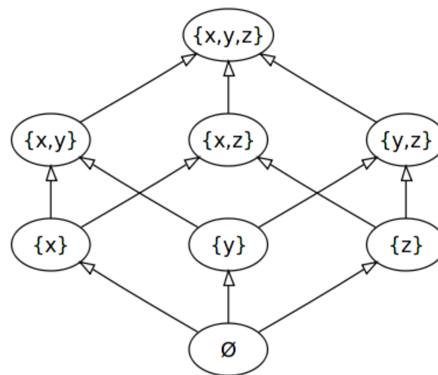


Figura 3.3.: Powerset con inclusione con 3 elementi

Esempio 3.30 (Funzioni parziali)
Prendiamo l'insieme di tutte le funzioni parziali sui naturali:

$$P = \omega \rightharpoonup \omega$$

Alternativamente possiamo vederlo come un insieme di relazioni ristrette a ritornare un solo valore (cioè al comportamento di una funzione).

$$P = \{R \subseteq \omega \times \omega| \quad nRm \quad nRm' \Longrightarrow m = m'\}$$

Notare che sono sempre parziali perché non è imposto che esista sempre m.
In questa visione le funzioni sono insiemi e quindi posso ordinarle per inclusione:

$$f \sqsubseteq g \Longleftrightarrow \forall x.(f(x) \text{ indefinita oppure } f(x) = g(x))$$

In altre parole, una funzione f è minore nell'ordinamento rispetto ad un'altra funzione g se esistono punti in cui g è definita ed f no (e non viceversa, ovviamente: in quel caso le due funzioni non sono in relazione).

Il $\bot$ di questo insieme è la funzione indefinita ovunque, cioè $\emptyset$, mentre esistono più upper bounds, cioè le funzioni totali.

Un altro modo per definire P è quello di aggiungere al codominio l'elemento indefinito $\bot$ e usare solo funzioni totali: $P = \omega \to \omega_\bot$. L'ordinamento diventa

$$f \sqsubseteq g \Leftrightarrow \forall x. f(x) \sqsubseteq_{\omega_\bot} g(x)$$

cioè se $f(x) = \bot \quad \Rightarrow \quad g(x)$ può assumere qualunque valore, l'elemento $\bot$ è la funzione che restituisce $\bot$ su qualunque ingresso e l'upper bound è la funzione per cui $\nexists i. f(i) = \bot$. Questa definizione ha il vantaggio che l'ordinamento sulle funzioni è demandato all'ordinamento sul codominio, che è: $\bot \sqsubseteq n$.

Rimane da dimostrare che l'ordinamento sia parziale completo.

Dimostrazione. *Se $(D, \sqsubseteq)$ è un ordinamento parziale, allora lo stesso ordinamento vale anche per ogni sottoinsieme $(D', \sqsubseteq)$. La completezza invece non è assicurata: devo mostrare che per ogni sottoinsieme esiste un lub.*

Presa una catena $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots$ con $\forall i. \quad nR_im \quad nR_im' \implies m = m'$, devo mostrare che la proprietà vale ancora per il limite della catena

$$n(\cup R_i)m \quad n(\cup R_i)m' \implies m = m'$$

Dove il limite è definito come $\cup R_i = \{(n, m) | \exists i. nR_im\}$.

Assumendo la premessa abbiamo che $\exists k, k'. \quad nR_km \quad nR_{k'}m'$, quindi se prendiamo un $k'' = max\{k, k'\}$ allora per entrabi vale la relazione $nR_{k''}m \quad nR_{k''}m'$ che per ipotesi implica $m = m'$. ☠

Esempio 3.31 (Sequenza di funzioni)
Definiamo una sequenza di funzioni $\{f_i\}_{i \in \omega}$ tale che

$$f_k(n) = \begin{cases} 3 \text{ se } n \text{ è pari } \wedge n \leq k \\ \bot \text{ altrimenti} \end{cases}$$

Vediamo qualche applicazione di esempio.

$$\begin{array}{lll} f_0(0) = 3 & & \\ f_1(0) = 3 & & \\ f_2(0) = 3 & f_2(2) = 3 & \\ f_3(0) = 3 & f_3(2) = 3 & \\ f_4(0) = 3 & f_4(2) = 3 & f_4(4) = 3 \end{array}$$

Ogni passo della catena è più definito del precedente. Il limite è la funzione definita su tutti i pari e indefinita altrove; anche il limite quindi è una funzione parziale.

3.1.5. Reticoli

Spesso in luogo dei PO si preferisce utilizzare delle strutture che prevedono delle proprietà più forti detti reticoli.

Definizione 3.32 (Reticolo)
Un reticolo è un insieme parzialmente ordinato $(P, \sqsubseteq)$ tale che per ogni sottoinsieme costituito da due elementi di P esistono lub e glb.

Vediamo un esempio di reticolo.

Esempio 3.33 (Reticolo)
Se prendiamo l'insieme

$$(\omega \cup \{\bot, \top\}, \leq)$$

avremo che $\forall n \in \omega : \bot \sqsubseteq n \sqsubseteq \top$*, per definizione di* $\top$ *e* $\bot$*. Qualunque coppia di elementi in* ω *si prenda, avranno chiaramente* $\top$ *come lub e* $\bot$ *come glb.*

Estendiamo la definizione individuando un sottoinsieme dei reticoli che ha proprietà molto più interessanti.

Definizione 3.34 (Reticolo completo)
Un insieme parzialmente ordinato $(P, \sqsubseteq)$ *è un* reticolo completo *se* $\forall X \subseteq P$ *esistono lub(X) e glb(X). In questo caso*

- $lub(P) = \top$ *è il* massimo *del reticolo*
- $glb(P) = \bot$ *è il* minimo *del reticolo*

Il reticolo completo si differenzia dal reticolo normale per il fatto che viene richiesta l'esistenza di lub e glb non solo per le coppie di elementi, ma per *tutti* i sottoinsiemi. è una proprietà molto più forte, ma che ci permetterà di fare ipotesi a loro volta molto più forti su tali insiemi.

Notare che ogni reticolo finito è un reticolo completo e, come vedremo nella sezione 3.1.4, ogni reticolo completo è un CPO. Vediamo un esempio di reticolo non completo.

Esempio 3.35 (Reticolo non completo)
$(\omega, \leq)$ *è un reticolo, ma non è completo in quanto ad esempio l'insieme dei numeri pari non ha lub.*

3.2. Verso la teoria del punto fisso

3.2.1. Monotonia e continuità

Ora che abbiamo definito correttamente un CPO $(D, \sqsubseteq)$, vogliamo definirci sopra delle funzioni che abbiamo un *punto fisso*: ricerchiamo dunque, per ogni F, un valore d tale che

$$d = F(d) \quad d \in D$$

Vedremo che trovare un punto fisso di una funzione rappresenterà il problema principale per la ricerca di una soluzione ai nostri problemi.

Troveremo che potremo affermare con certezza che esiste un punto fisso solo per certe classi di funzioni, e che solo in certi casi avremo una procedura che lo calcoli in tempo finito.

Vediamo inizialmente di definire due proprietà sulle nostre funzioni che ci aiuteranno a comprenderne la forma. Per prima cosa introduciamo una notazione nuova.

Notazione
Con la notazione $\bigsqcup_{i \in \omega} \{a_i\}$ *si intende il limite della successione* $\{a_i\}$ *per i sempre più grande.*

Definiamo adesso la prima proprietà interessante, cioè la *monotonia*.

Definizione 3.36 (Monotonia)
Una funzione F su un CPO $(D, \sqsubseteq)$ *è monotona se e solo se*

$$\forall d, e \in D . d \sqsubseteq e \Rightarrow F(d) \sqsubseteq F(e)$$

La funzione in pratica deve mantenere la struttura del dominio. Questo ci porta anche a dire che se d_i è una catena e F è monotona allora anche $F(d_i)$ è una catena, in quanto l'ordinamento definito sul dominio di partenza non viene variato dalla funzione.

Vediamo un esempio.

Esempio 3.37 (Funzione non monotona)
Esempio di funzione non monotona su un CPO con $b \sqsubseteq a$, $b \sqsubseteq c$:

$$f(a) = a \quad f(b) = a \quad f(c) = c$$

Questa funzione modifica l'ordinamento: non vale $b \sqsubseteq c \not\Rightarrow f(b) \sqsubseteq f(c)$ in quanto a e c non sono in relazione.

Vediamo adesso la seconda proprietà interessante, la *continuità*.

Definizione 3.38 (Continuità)
Una funzione F su un CPO $(D, \sqsubseteq)$ è continua sse per ogni catena d definita su D si ha

$$F(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} F(d_i)$$

In altre parole, presa una funzione F questa è continua se la funzione applicata al limite del CPO restituisce un risultato uguale al limite del CPO ottenuto applicando la funzione a tutti gli elementi del CPO di partenza.

Esempio 3.39 (Funzione monotona non continua)
Dato $(\omega \cup \{\infty\}, \leq)$ CPO, definiamo la funzione

$$\begin{cases} f(n) = 0 \\ f(\infty) = 1 \end{cases}$$

Scelta una catena del tipo

$$0 \leq 2 \leq 4 \leq 6 \leq \dots$$

Abbiamo che

$$f(\bigsqcup d_i) = f(\infty) = 1 \quad \neq \quad \bigsqcup f(d_i) = 0$$

La funzione non è quindi continua.

Queste due nozioni sono in forte relazione fra di loro, come possiamo vedere dalle seguenti proprietà.

Cominciamo a limitare il nostro campo capendo come si comportano queste due proprietà con catene finite.

Teorema 3.40 (Relazione fra monotonia e continuità (1))
Se una catena $\{d_i\}$ è finita allora ogni funzione monotona definita su tale catena è continua.

Dimostrazione. Data

$$d_0 \sqsubseteq d_1 \sqsubseteq \dots \sqsubseteq d_k \sqsubseteq d_k \sqsubseteq \dots$$

questa sarà una catena finita con limite $\bigsqcup d_i = d_k$ (ricordiamo che una catena finita è considerata come una catena infinita nella quale dopo un numero finito di passi tutti gli elementi sono uguali).

Visto che f è monotona, possiamo concludere che la catena

$$f(d_0) \sqsubseteq f(d_1) \sqsubseteq \ldots \sqsubseteq f(d_k) \sqsubseteq f(d_k) \sqsubseteq \ldots$$

si stabilizza su $f(d_k)$, che sarà dunque limite della catena risultante. Quindi

$$f(\bigsqcup_{i \in \omega} d_i) = \bigsqcup_{i \in \omega} f(d_i)$$

ed ovviamente la funzione è continua. ☠

Il seguente teorema non verrà dimostrato.

Teorema 3.41 (Relazione fra monotonia e continuità (2))
Una funzione continua è anche monotona

3.2.2. Punto fisso

Possiamo finalmente definire formalmente cosa è un punto fisso.

Definizione 3.42 (Punto fisso)
Sia f una funzione continua su un $CPO_\perp$. Un elemento d si dice punto fisso se e solo se vale:

$$f(d) = d$$

L'insieme dei punti fissi è denotato con $Fix(f)$.

Definiamo anche un altro importante insieme di elementi che costituiscono la strada verso il nostro punto fisso.

Definizione 3.43 (Punto prefisso)
Sia f una funzione continua su un $CPO_\perp$. Un elemento d si dice punto prefisso se e solo se vale:

$$f(d) \sqsubseteq d$$

Data una funzione f, indicheremo con $gfp(f)$ (greatest fixed point) il suo massimo punto fisso, e con $lfp(f)$ (least fixed point) il minimo punto fisso.

Abbiamo visto cos'è un punto fisso, ma le domande più pressanti sono quando questo punto esiste e come possiamo trovarlo. Daremo due risposte a questa domanda

- il teorema del punto fisso o di Kleene
- il teorema di Tarski

3.2.3. Il teorema di Kleene

Questa prima caratterizzazione del punto fisso è dovuta a Kleene, che ha costruito questo teorema. Vediamo prima questo banale lemma che ci servirà per la dimostrazione

Lemma 3.44 (Limite indipendente dall'inizio della catena)
Due catene

$$\begin{matrix} a_0 \sqsubseteq a_1 = f(a_0) \sqsubseteq a_2 = f(a_1) \sqsubseteq \dots \\ a_i \sqsubseteq a_{i+1} = f(a_i) \sqsubseteq a_{i+2} = f(a_{i+1}) \sqsubseteq \dots \end{matrix} \Rightarrow \text{ stesso limite!}$$

nelle quali un elemento e' ottenuto in funzione del precedente mantengono uguale il valore del limite togliendo un numero finito di elementi dalla testa della catena.

Grazie a questo lemma, possiamo dimostrare il seguente teorema.

Teorema 3.45 (Teorema del punto fisso (o teorema di Kleene))
Sia $f : D \to D$ una funzione continua su un ordinamento parziale completo con elemento bottom $CPO_\perp$. Allora esiste un valore

$$fix(f) = \bigsqcup_{n \in \omega} f^n(\perp)$$

tale che

1. *fix(f) è punto fisso di f*
$$f(fix(f)) = fix(f)$$
2. *fix(f) è il minimo punto prefisso di f*
$$f(d) \sqsubseteq d \Rightarrow fix(f) \sqsubseteq d$$

Dunque, in particolare, $fix(x)$ è il minimo punto fisso di f, cioè $fix(f) = lfp(f)$.

Dimostrazione. Per prima cosa, vediamo che $f^n(\perp)$ è una catena. Abbiamo

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp) \sqsubseteq \dots$$

Definendola per costruzione possiamo dire

$$f^{n+1}(x) = f(f^n(x)) \qquad f^0(x) = x$$

Per induzione vediamo che

- $f^0(\perp) = \perp$ è sicuramente il minimo, in quanto $\perp$ è minore di qualunque altra cosa
- Dobbiamo dimostrare il passo induttivo, e cioè che $f^n(\perp) \sqsubseteq f^{n+q}(\perp)$. Sappiamo per ipotesi induttiva che $f^{n-1}(\perp) \sqsubseteq f^n(\perp)$, e che f è monotona: dunque possiamo concludere $f^{n-1}(\perp) \sqsubseteq f^n(\perp) \Rightarrow f(f^{n-1}(\perp)) \sqsubseteq f(f^n(\perp))$, che e' per definizione equivalente a quanto vogliamo dimostrare.

Siamo dunque certi che il limite esiste, in quanto per la definizione di CPO che troviamo a pagina 38 una catena ha sempre un limite. Sulla base di questo, dimostriamo le due affermazioni

1. Dobbiamo dimostrare che fix(f) è un punto fisso di f. Questo è facilmente dimostrabile
   - Per definizione abbiamo
   $$f(fix(f)) = f(\bigsqcup_{n \in \omega} f^n(\perp))$$
   - Visto che f è continua possiamo riscrivere come
   $$f(\bigsqcup_{n \in \omega} f^n(\perp)) = \bigsqcup_{n \in \omega} f(f^n(\perp)) = \bigsqcup_{n \in \omega} f^{n+1}(\perp)$$

- per il lemma 3.44 abbiamo che, essendo $f^{n+1}(\bot)$ uguale a $f^n(\bot)$ dalla quale è stato tolto il primo elemento,

$$\bigsqcup_{n\in\omega} f^{n+1}(\bot) = \bigsqcup_{n\in\omega} f^n(\bot) = fix(f)$$

Riunendo le uguaglianze, abbiamo che $f(fix(f)) = fix(f)$, e che dunque fix(f) è punto fisso per f.

2. Assumiamo l'esistenza di un punto d tale che $f(d) \sqsubseteq d$ (d è ovviamente un punto prefisso per f). Vogliamo dimostrare per induzione che $fix(f) = \bigsqcup_{i\in\omega} f^n(\bot) \sqsubseteq d$. Vediamolo per induzione.

Caso base banale: per definizione $f^0(\bot) = \bot$ e $\bot \sqsubseteq d$.

Caso induttivo Al solito, assumiamo $f^n(\bot) \sqsubseteq d$ e dimostriamo $f^{n+1}(\bot) \sqsubseteq d$. Anche questo passo è semplice:

$$\begin{aligned} f^{n+1}(\bot) = f(f^n(\bot)) \sqsubseteq & \quad \text{per definizione} \\ \sqsubseteq f(d) \sqsubseteq & \quad \text{per la monotonia di f} \\ \sqsubseteq d & \quad \text{per ipotesi} \end{aligned}$$

Ne concludiamo che fix(f) è minimo punto prefisso di f.

L'affermazione è dunque completamente dimostrata: fix(f) è allo stesso tempo punto fisso e minimo punto prefisso, quindi è anche minimo punto fisso. ☠

La continuità della funzione e la presenza dell'elemento $\bot$ nell'insieme sono necessari per la presenza del lfp. Dimostriamolo attraverso due esempi.

Esempio 3.46 ($\bot$ necessario)
Prendiamo ({True,False},Identità), è completo, ha monotonia, ha punti fissi.
I punti fissi però sono due, nessuno minimo. Quindi, perché vi sia un lfp, è necessaria la presenza dell'elemento $\bot$.

Esempio 3.47 (Continuità necessaria)
Prendiamo come cpo $\omega \cup \{\infty_1, \infty_2\}$ con $x \leq \infty$ e f:

$$f(n) = n + 1 \quad f(\infty_1) = \infty_2 \quad f(\infty_2) = \infty_2$$

abbiamo che f non è continua

$$\bigsqcup d_i = \infty_1 \quad f(\bigsqcup d_i) = \infty_2 \quad \bigsqcup f(d_i) = \infty_1$$

Prendiamo la catena dei numeri primi

$$2 \leq 3 \leq 5 \leq 7 \leq \dots$$

Il punto fisso di f è per definizione ∞_2 ma

$$\begin{array}{llll} fix(f) = \bigsqcup f^n(\bot)) = \bot & \sqsubseteq f(\bot) & \sqsubseteq f(f(\bot)) & \sqsubseteq \dots \\ = 0 & \sqsubseteq 1 & \sqsubseteq 2 & \sqsubseteq \dots = \infty_1 \end{array}$$

quindi non si riesce a raggiungere il punto fisso con il metodo iterativo.

3.2.4. Il teorema di Tarski

Il teorema di Tarski da una differente interpretazione del punto fisso, trovandolo a partire da una funzione monotona (e NON necessariamente continua!) e da un reticolo completo.

Teorema 3.48 (Teorema di Tarski)
Sia f una funzione monotona su un reticolo completo $(D, \sqsubseteq)$. Allora f ha un minimo punto fisso e in particolare

$$lfp(f) = glb(\{x \in D | f(x) \sqsubseteq x\})$$

Dimostrazione. Prendiamo $X = \{x \in D | f(x) \sqsubseteq x\}$, e prendiamo m = glb(X).

- Preso un qualunque $x \in X$ avremo ovviamente che $m \sqsubseteq x$ per definizione del glb.
- Per la monotonia di f avremo dunque che $f(m) \sqsubseteq f(x)$, e visto che $x \in X$ avremo che $\forall x \in X. f(x) \sqsubseteq x \Rightarrow f(m) \sqsubseteq x$.
- Quindi siamo sicuri che $f(m) \sqsubseteq glb(X) = m$, dunque m è punto prefisso e per sua definizione è il minimo dei punti prefissi.
- Per la monotonia di f, $f(m) \sqsubseteq m \Rightarrow f(f(m) \sqsubseteq f(m)$, il che implica che $f(m) \in X$ e che dunque $m \sqsubseteq f(m)$
- Ma ovviamente $m \sqsubseteq f(m) \wedge f(m) \sqsubseteq m \Rightarrow m = f(m)$:

Si conclude che m è sia punto fisso che minimo punto prefisso, ed è dunque minimo punto fisso. ☠

Quindi abbiamo due risultati che garantiscono l'esistenza del minimo punto fisso:

- funzione monotona su CPO
- funziona continua su CPO

Il primo garantisce solo l'esistenza del lfp mentre il secondo ha ipotesi più forti, e fornisce un metodo per calcolarlo.

3.3. Operatore delle conseguenze immediate

3.3.1. L'operatore $\hat{R}$

Quanto visto in questo capitolo è una branca importante della logica e della matematica, ma dobbiamo ancora vedere come possiamo applicare questi concetti al nostro campo di studi. Vediamo adesso un'applicazione che ci tornerà utilissima nei prossimi capitoli, dove vedremo la semantica denotazionale di IMP e gli altri linguaggi.

Immaginiamo di avere un sistema di regole di inferenza, e si consideri come dominio nel quale operiamo l'insieme delle formule ben formate $\mathscr{F}$ di questo sistema di regole. Se prendiamo l'insieme $2^{\mathscr{F}}$ tutti i sottoinsiemi di $\mathscr{F}$ e la relazione di sottoinsieme, questi formano un $CPO_{\perp}$ come visto nell'esempio 3.29.

Vogliamo mostrare che, una volta definito un particolare operatore su tale CPO, l'insieme di teoremi I_R di un certo insieme di regole R è un punto fisso di tale operatore. Questo operatore si chiama *operatore delle conseguenze immediate*, e lo andiamo a definire immediatamente.

Definizione 3.49 (Operatore delle conseguenze immediate ($\hat{R}$))
Dati un insieme di regole R e un insieme $B \in \mathscr{F}$, l'operatore delle consegueze immediate è definito come

$$\hat{R}(B) = \left\{ y | \exists (^X/_y) \in R. X \subseteq B \right\}$$

In altre parole, $\hat{R}(B)$ è l'insieme delle *conseguenze immediate* di B, cioè delle formule ben formate che è possibile dedurre dalle regole contenute in R e dall'insieme delle fbf contenute in B: una formula y appartiene a $\hat{R}(B)$ se appare come conseguenza in una regola della quale tutte le premesse sono contenute in B. L'operatore tenta di applicare tutte le regole nei possibili modi per derivare le conseguenze di B e sotto alcune condizioni ha punto fisso.

Immaginiamo adesso di costruire una successione di sottoinsiemi nel CPO che stiamo studiando così definita

- il $\perp$ della catena è l'insieme vuoto
- ogni elemento della catena è ottenuto applicando all'elemento precedente $\hat{R}$

Guardando la sequenza di elementi della catena, notiamo che a qualunque passo otteniamo un sovrainsieme dell'insieme precedente. Dimostriamolo.

Teorema 3.50 (Catena di applicazioni $\hat{R}(\perp)$)
Se costruisco una successione $\{A_i\}_{i\in\omega}$ di sottoinsiemi di formule ben formate $\mathscr{F}$, nella quale

- $A_0 = \varnothing$
- $A_{n+1} = \hat{R}(A_n)$

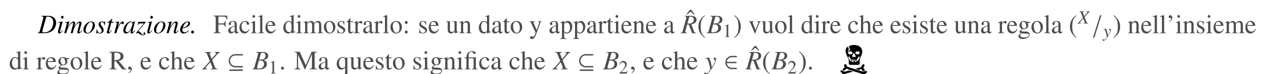
allora è una catena sul CPO $\left(2^{\mathscr{F}}, \subseteq\right)$

Dimostrazione. Dimostriamolo per induzione:

1. Dobbiamo dimostrare che $A_0 \subseteq A_1$. è facile farlo: A_0 è l'insieme vuoto, quindi chiunque sia A_1 sarà sovrainsieme dell'inseme vuoto.
2. Dobbiamo dimostrare che $A_n \subseteq A_{n+1}$ o, in altre parole, $y \in A_n \Rightarrow y \in A_{n+1}$.
   Per ipotesi induttiva supponiamo che $z \in A_{n-1} \Rightarrow z \in A_n$. Quindi, visto che $y \in A_n \Rightarrow \exists X \subseteq A_{n-1}.\left({}^{X}/_{y}\right) \in R$, avremo che $X \subseteq A_{n-1} \subseteq A_n$, e che dunque $X \subseteq A_n$. Ma ovviamente questo, assieme all'esistenza della regola $\left({}^{X}/_{y}\right) \in R$, implicano che y debba essere presente anche in A_{n+1}!

Cominciamo a studiare alcune proprietà del nostro operatore $\hat{R}(B)$.

Teorema 3.51 (Monotonia di $\hat{R}$)
L'operatore $\hat{R}$ è monotono.


Dimostrazione. Facile dimostrarlo: se un dato y appartiene a $\hat{R}(B_1)$ vuol dire che esiste una regola $({}^{X}/_{y})$ nell'insieme di regole R, e che $X \subseteq B_1$. Ma questo significa che $X \subseteq B_2$, e che $y \in \hat{R}(B_2)$. ☠

Teorema 3.52 (Continuità di $\hat{R}$)
Se R è un insieme di regole di inferenza in cui tutte le regole hanno un numero finito di premesse, l'operatore $\hat{R}$ è continuo.

Dimostrazione. Se $\hat{R}$ è continuo, vuol dire che $\bigcup_{n\in\omega}\hat{R}(B_n) = \hat{R}(\bigcup_{n\in\omega} B_n)$[2]. Possiamo spezzare questa affermazione in due parti, sfruttando la proprietà antisimmetrica. Dobbiamo dunque dimostrare che

1. $\bigcup_{n\in\omega}\hat{R}(B_n) \subseteq \hat{R}(\bigcup_{n\in\omega} B_n)$
2. $\bigcup_{n\in\omega}\hat{R}(B_n) \supseteq \hat{R}(\bigcup_{n\in\omega} B_n)$

Vediamole separatamente.

1. Dobbiamo dimostrare che $\bigcup_{n\in\omega}\hat{R}(B_n) \subseteq \hat{R}(\bigcup_{n\in\omega} B_n)$

   Prendiamo una generica catena $\{B_n\}$ nel nostro insieme delle parti dell'insieme delle formule ben formate di R. Visto che il nostro insieme è un $CPO_\perp$, questa catena avrà lub che corrisponderà al limite della catena stessa. Per definizione dunque avremo che $B_n \subseteq \bigcup_{n\in\omega} B_n$, e poiché $\hat{R}$ è monotona varrà anche $\hat{R}(B_n) \subseteq \hat{R}\left(\bigcup_{n\in\omega} B_n\right)$. Anche questa sarà una catena, che avrà $\hat{R}\left(\bigcup_{n\in\omega} B_n\right)$ come maggiorante. Visto che per definizione $\bigcup_{n\in\omega}\hat{R}(B_n)$ è per definizione il minimo dei maggioranti della catena, avremo sicuramente la formula da dimostrare come conseguenza.

2. Dobbiamo dimostrare che $\bigcup_{n\in\omega}\hat{R}(B_n) \supseteq \hat{R}(\bigcup_{n\in\omega} B_n)$. Questo è equivalente a dire

   $$\forall y. y \in \hat{R}(\bigcup B_n) \Rightarrow y \in \bigcup \hat{R}(B_n)$$

   Assumiamo la premessa. Se X è un insieme infinito possiamo fare poco. Ma possiamo vivere tranquillamente nell'assunzione che X sia finito (e che dunque ogni regola abbia un numero finito di premesse): in questo caso $\exists k. X \in B_k$, in caso contrario X non apparterrebbe neppure al limite!

   Ma allora $y \in \hat{R}(B_k)$. Questo significa ovviamente che $y \in \bigcup_{n\in\omega}\hat{R}(B_n)$, come volevasi dimostrare.

Dalle due formule dimostrate, per la proprietà di antisimmetria di $\subseteq$ possiamo concludere che i due insiemi sono uguali e che dunque $\hat{R}$ è completa.

3.3.2. Punto fisso di $\hat{R}$

A questo punto, abbiamo tutti i tasselli. Abbiamo un ordinamento parziale con bottom sul quale è stato definito un operatore continuo, quindi è possibile usare il Teorema di Kleene per trovare il punto fisso.

In questo caso l'insieme dei teoremi I_R è l'insieme definito come

$$I_R = \bigcup_{i\in\omega} I_R^i$$

dove

$$I_R^0 \stackrel{\text{def}}{=} \emptyset$$
$$I_R^{n+1} \stackrel{\text{def}}{=} \left\{y | \exists \left({}^X/_y\right) \in R. X \subseteq I_R^n\right\} \cup I_R^n$$

In parole povere, I_R^n contiene tutti i teoremi definibili tramite n derivazioni, ed I_R contiene tutti i teoremi definibili. Vediamo adesso che I^R è punto fisso di $\hat{R}$.

Teorema 3.53 (Punto fisso di $\hat{R}$)
Dato un insieme R, vale

$$fix(\hat{R}) = I_R$$

[2] notare che utilizziamo il simbolo $\cup$ per indicare il limite: essendo ogni catena composta da elementi uno sottoinsieme dell'altro, il limite della catena non è altro che l'unione di tutti i suoi elementi

Dimostrazione. É facile vedere che $I_R^{n+1} = \left\{ y | \exists \left({}^X/_y \right) \in R.X \subseteq I_R^n \right\} \cup I_R^n = \hat{R}(I_R^n) \cup I_R^n = \hat{R}(I_R^n)$. Quindi la successione $\bigcup_{n\in\omega} I_R^n$ è la stessa vista nel teorema 3.50.

Essendo $\hat{R}$ una funzione continua su un $CPO_\perp$, per il teorema del punto fisso abbiamo

$$fix\hat{R} = \bigcup_{n\in\omega} S_n = \bigcup_{n\in\omega} I_R^n = I_R$$

dove $S_n = \hat{R}^n(\varnothing)$

$$S^0 = \varnothing$$
$$S^{n+1} = \hat{R}(S^n)$$

Mostriamo prima che $S^{n+1} \supseteq S^n$ per induzione matematica

0) $S^1 \supseteq \perp$

n) $S^{n+1} \supseteq S^n \Rightarrow S^{n+2} \supseteq S^{n+1}$ per monotonia di $\hat{R}$.

Vogliamo ora mostrare per induzione matematica che

$$S^n \stackrel{?}{=} I_R^n$$

0) In 0 sono entrambe $\varnothing$.

n) Dobbiamo mostrare che

$$I_R^n = S_n \quad \Rightarrow \quad I_R^{n+1} = S_{n+1}$$

Assumiamo la premessa e usiamo il risultato della precedente dimostrazione

$$\begin{aligned} I_R^{n+1} &= \hat{R}(I_R^n) \cup I_R^n \\ &= \hat{R}(S^n) \cup S^n \\ &= S^{n+1} \cup S^n = S^{n+1} \end{aligned}$$

Da cui abbiamo che

$$\forall n. S_n = I_R^n$$

quindi

$$I_R = \bigcup_{n\in\omega} I_R^n = \bigcup_{n\in\omega} S^n \stackrel{\text{def}}{=} fix(\hat{R})$$

Esempio 3.54 (Insieme di regole con $\hat{R}$ non continuo)

$$P(1) \qquad \frac{P(x)}{P(x+1)} \qquad \frac{\forall x \textit{ dispari } P(x)}{P(0)}$$

Notare che la terza regola ha infinite premesse.

Qui non è vero che $\forall \{B_n\}_{n\in\omega} \quad \cup_{n\in\omega} \hat{R}(B_n) = \hat{R}(\cup_{n\in\omega} B_n)$ *Scegliamo la catena di tutti i numeri dispari*

$$\{P(1)\} \quad \{P(1), P(3)\} \quad \{P(1), P(3), P(5)\} \ldots$$

Abbiamo costruito una catena che è buona nell'ordinamento che abbiamo.
Se vado a vedere $\hat{R}$ nel generico elemento della sequenza, ossia $\hat{R}(S_i)$

S_i	$\{P(1)\}$	$\{P(1), P(3)\}$	$\{P(1), P(3), P(5)\}$
$\hat{R}(S_1)$	$\{P(1), P(2)\}$	$\{P(1), P(2), P(4)\}$	$\{P(1), P(2), P(4), P(6)\}$

$$\bigcup S_i = \{P(1), P(3), P(5), \ldots\}$$
$$\bigcup \hat{R}(S_i) = \{P(1), P(2), P(4), \ldots\}$$

$$\hat{R}(\bigcup S_i) = \{P(1), P(2), P(4) \ldots \underbrace{P(0)}_{\text{3a regola}}\}$$

La terza regola si applica solo quando sono presenti tutti i dispari, cioè al limite.

Non è continua, ovvero non vale $\bigcup\{\hat{R}(s_i)\} = \hat{R}(\bigcup\{s_i\})$.

$$\emptyset \subseteq \hat{R}(\emptyset) \subseteq \ldots$$
$$\{\} \subseteq \{P(1)\} \subseteq \{P(1), P(2)\} \subseteq \ldots$$

Di conseguenza, non si può applicare il Teorema del Punto fisso (le sue ipotesi non sono verificate).

$$fix(\hat{R}) = \bigcup_{n\in\omega} R^n(\emptyset) = \{P(1), P(2), \ldots\}$$
$$\hat{R}(fix(\hat{R})) = \{P(0), P(1), P(2) \ldots\}$$

Il teorema del punto fisso non vale perché facendo $fix(\hat{R})$ *non troviamo il punto fisso, che invece è* $\hat{R}(fix(\hat{R}))$.

Esempio 3.55 (Stringhe di una grammatica)
Grammatica

```
S::= lambda | (S) | SS
```

$$S_0 = \emptyset$$
$$S_1 = \lambda + (\emptyset) + \emptyset\emptyset = \lambda$$
$$S_2 = \lambda + (\lambda) + \lambda\lambda = \lambda + ()$$
$$S_3 = \lambda + () + (()) + ()()$$

4. Semantica denotazionale di IMP

Definire la semantica di un linguaggio di programmazione è un lavoro che può essere svolto in maniere completamente differenti fra di loro. Introduciamo adesso un secondo formalismo per definire la semantica del nostro linguaggio IMP: la semantica *denotazionale*.

Se la semantica operazionale è un formalismo che concettualmente si avvicina al concetto di interprete, in quanto prende in ingresso un programma ed uno stato e restituisce il risultato della computazione su tale stato, la semantica denotazionale si avvicina più al concetto di compilatore. Infatti, prende in ingresso il solo programma e restituisce una funzione (ovvero un programma scritto in un altro formalismo) che prende in ingresso lo stato iniziale e restituisce lo stato finale.

Per farlo, ci servirà un *metalinguaggio* nel quale esprimere questa funzione. Questo linguaggio è il **λ-calcolo**, che definiremo nella sezione 5.1.1.

4.1. Semantica denotazionale di IMP

Come abbiamo visto nell'introduzione, la semantica è una funzione che prende un programma e ne restituisce il significato, o denotazione. Abbiamo anche visto che la semantica operazionale lega l'interpretazione ad uno stato che deve essere fornito assieme al programma mentre la semantica denotazionale fornisce una funzione alla quale si passa lo stato per vedersi fornire un risultato.

La nostra semantica per IMP sarà in realtà divisa in tre funzioni separate, una per ogni categoria sintattica di IMP. Avremo dunque le funzioni

- $\mathscr{A} : Aexpr \to \Sigma \to \mathbb{N}$ che prende in ingresso un'espressione aritmetica e restituisce una funzione da stati ad interi
- $\mathscr{B} : Bexpr \to \Sigma \to \mathbb{B}$ che prende in ingresso un'espressione booleana e restituisce una funzione da stati a boolean
- $\mathscr{C} : Com \to \Sigma \to \Sigma_\perp$ oppure $Com \to \Sigma \rightharpoonup \Sigma$, che prende in ingresso un comando e restituisce una funzione da stati a stati

Vediamo separatamente queste tre funzioni.

4.1.1. Funzione $\mathscr{A}$

Vogliamo definire la funzione di interpretazione semantica relativa alle espressioni aritmetiche

$$\mathscr{A} : Aexpr \to \Sigma \to \mathbb{N}$$

La definiremo per ricorsione strutturale, lavorando come nel caso della semantica operazionale sulla struttura della grammatica che definisce le espressioni in Aexpr. Vediamo la notazione.

Notazione
Nell'espressione

$$\mathscr{A}\,[\![n]\!] = \lambda\sigma.n$$

abbiamo che

- *$\mathscr{A}$ è, come detto precedentemente, una funzione $Aexpr \to \Sigma \to \mathbb{N}$*
- *n è una generica Aexpr. In generale dentro le parentesi $[\![\ e\ ]\!]$ andranno espressioni in Aexpr*

- *$\mathscr{A}\,[\![n]\!]$ è una funzione $\Sigma \to \mathbb{N}$, così come $\lambda\sigma.n$*
- *la notazione $\mathscr{A}\,[\![n]\!]\,\sigma$ indicherà l'applicazione di $\mathscr{A}\,[\![n]\!]$ all'argomento σ, ottenendo ovviamente un elemento di $\mathbb{N}$.*

Solitamente utilizzeremo uguaglianze del tipo

$$\mathscr{A}\,[\![n]\!]\,\sigma = n$$

per evitarci di scrivere ogni volta il $\lambda\sigma$ all'inizio dell'espressione, anche se formalmente sarebbe più corretto scrivere

$$\mathscr{A}\,[\![n]\!] = \lambda\sigma.n$$

Possiamo passare a definire le regole per $\mathscr{A}$.

Costanti Per le costanti la regola è semplicissima.

$$\mathscr{A}\,[\![n]\!]\,\sigma = n$$

La semantica di una costante n è semplicemente il corrispondente valore numerico.

Variabili Per le variabili la regola è ancora banale.

$$\mathscr{A}\,[\![x]\!]\,\sigma = \sigma x$$

Si tratta dell'applicazione della funzione ambiente, che ricordiamo essere una funzione da identificatori a valori numerici, all'identificatore x.

Espressioni binarie Per le espressioni binarie la regola dice

$$\mathscr{A}\,[\![a_0 + a_1]\!]\,\sigma = \mathscr{A}\,[\![a_0]\!]\,\sigma \underline{+} \mathscr{A}\,[\![a_1]\!]\,\sigma$$

Stesso discorso possiamo fare per sottrazione e moltiplicazione.

Abbiamo dunque dato la definizione di $\mathscr{A}$ per ricorsione strutturale.

4.1.2. Funzione $\mathscr{B}$

La funzione $\mathscr{B}$ è sostanzialmente analoga alla precedente, ad eccezione ovviamente del tipo di dato restituito che è $\mathbb{B}$ invece che $\mathbb{N}$. I discorsi fatti sui tipi delle espressioni in gioco sono analoghi a quelli fatti precedentemente, così come le regole di transizione. Vediamo solo brevemente, per curiosità, l'unica che si distacca leggermente da quanto visto con $\mathscr{A}$.

$$\mathscr{B}\,[\![\neg b]\!]\,\sigma = \neg\mathscr{B}\,[\![b]\!]\,\sigma$$

Al solito, i due $\neg$ sono simboli uguali ma hanno un significato differente.

4.1.3. Funzione $\mathscr{C}$

La funzione $\mathscr{C}$, al solito, si differenzia abbastanza dall'altra gamma di funzioni. Se come sempre abbiamo i comandi banali come

$$\mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma$$

$$\mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma\left[{}^{\mathscr{A}[\![a]\!]}/_{x}\right]$$

che si risolvono facilmente, le cose possono facilmente complicarsi in questo caso.

$$\mathscr{C}\,[\![c_1; c_2]\!]\,\sigma = \mathscr{C}\,[\![c_2]\!]\,(\mathscr{C}\,[\![c_1]\!]\,\sigma)$$

Se la valutazione di $\mathscr{C}\,[\![c_1]\!]\,\sigma$ restituisce $\perp$ la situazione diventa complessa: come comportarsi in caso di computazioni che *non terminano*?. È facile trovare una soluzione. Basta definire, a partire da una qualunque funzione $f : \Sigma \to \Sigma_\perp$ una funzione $f^* : \Sigma_\perp \to \Sigma_\perp$ tale che

$$f^*(x) = \begin{cases} \perp & \text{se } x = \perp \\ f(x) & \text{altrimenti} \end{cases}$$

In pratica, se una funzione interna restituisce $\perp$ anche tutte le funzioni che utilizzeranno lo stato da essa prodotto restituiranno $\perp$. Nella sequenzializzazione di comandi è l'espressione $\mathscr{C}\,[\![c_1]\!]\,\sigma$, eseguita per prima, che potrebbe non terminare.

La versione riveduta della concatenazione è dunque

$$\mathscr{C}\,[\![c_1; c_2]\!]\,\sigma = \mathscr{C}\,[\![c_2]\!]^*\,(\mathscr{C}\,[\![c_1]\!]\,\sigma)$$

Torniamo alla semantica. La regola per gestire l'if è ancora più semplice di quella operazionale.

$$\mathscr{C}\,[\![\textbf{if}\; b\; \textbf{then}\; c_1\; \textbf{else}\; c_2]\!]\,\sigma = \mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c_1]\!]\,\sigma, \mathscr{C}\,[\![c_2]\!]\,\sigma$$

Si utilizza pari pari il costrutto di λ-calcolo che modella il condizionale. Passiamo all'unico caso complesso, cioè il while. Una definizione immediata ma *scorretta!* sarebbe

$$\mathscr{C}\,[\![\textbf{while}\; b\; \textbf{do}\; c]\!]\,\sigma = \mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![\textbf{while}\; b\; \textbf{do}\; c]\!]^*\,(\mathscr{C}\,[\![c]\!]\,\sigma)\,, \sigma$$

Ricordiamo che vogliamo dare la semantica denotazionale per ricorsione strutturale, in modo da ottenere la proprietà di *composizionalità*: ogni comando deve essere costruito dalla denotazioni dei suoi immediati sottocomandi. Questa definizione non ne segue i principi in quanto nella parte destra abbiamo il termine stesso.

Si noti che la proprietà di composizionalità non è presente nella semantica operazionale dove infatti la regola del while viene trattata facendone l'*unfolding* e dove il while riappare nelle premesse. Ritornando al paragone della semantica denotazionale con un compilatore vediamo come questo approccio non sia possibile perché nella traduzione del programma saremmo costretti ad eseguirlo senza una memoria di partenza.

Per risolvere il problema ci viene in aiuto il concetto di punto fisso. Prima di tutto riscriviamo la formula esplicitando il fatto che lo stato è un parametro della funzione λ-calcolo

$$\mathscr{C}\,[\![\textbf{while}\; b\; \textbf{do}\; c]\!] = \lambda\sigma.\mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![w]\!]^*\,(\mathscr{C}\,[\![c]\!]\,\sigma)\,, \sigma$$

A questo punto consideriamo una funzione così definita:

$$\Gamma = \underbrace{\lambda\varphi.\, \underbrace{\lambda\sigma.\, \underbrace{\mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}\,[\![c]\!]\,\sigma), \sigma}_{\Sigma_\perp}}_{\Sigma\to\Sigma_\perp}}_{(\Sigma\to\Sigma_\perp)\to\Sigma\to\Sigma_\perp}$$

La funzione Γ ha dunque tipo $(\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$. Di questa funzione dobbiamo cercare, ammesso che esista, un punto fisso. Naturalmente si può dimostrare che la Γ è continua.

Questa è una definizione valida: la semantica del while è dunque la minima φ che soddisfa $\varphi = \Gamma\varphi$ e cioè il *minimo punto fisso* della funzione Γ.

$$\mathscr{C}\,[\![\textbf{while}\; b\; \textbf{do}\; c]\!] = fix\;\Gamma$$

con

$$fix = \cup_{n\in\omega}(\lambda f.\; f_n(\perp))$$

Esempio 4.1 (Semantica denotazionale di un comando)
Consideriamo il comando

$$w = \textbf{while true do skip}$$

e calcoliamone la semantica.
$\mathscr{C}\,[\![w]\!] = \textbf{fix}\,\Gamma$ *dove*

$$\Gamma\varphi\sigma \quad = \quad \mathscr{B}\,[\![\textbf{true}]\!]\,\sigma \to \varphi^*\left(\mathscr{C}\,[\![\textbf{skip}]\!]\,\sigma\right), \sigma \quad = \quad \varphi^*\left(\mathscr{C}\,[\![\textbf{skip}]\!]\,\sigma\right) \quad = \quad \varphi^*\sigma \quad = \quad \varphi\sigma$$

Abbiamo dunque una situazione in cui deve valere $\Gamma\varphi = \varphi$, cioè l'identità. La formula vale per qualunque φ: è per questo che consideriamo il minimo punto fisso, che nell'occasione è ovviamente la funzione ovunque indefinita $\perp$. *In effetti, la semantica della funzione è proprio questa: la funzione è ovunque indefinita, cioè per qualunque memoria σ in input il programma non termina.*

Vediamo quel che abbiamo appena visto in via teorica. Dobbiamo appoggiarci alla teoria del punto fisso.
Il nostro dominio è $(\Sigma \to \Sigma_\perp, \sqsubseteq)$. Abbiamo già visto nell'esempio 3.30 che, considerando le funzioni come coppie in un insieme F $((a, b) \in F \Leftrightarrow f(a) = b)$, questo ordinamento è

- parziale
- completo
- ha un bottom

Se dimostriamo che Γ è monotona e continua, abbiamo anche dimostrato che ha minimo punto fisso. Per dimostrarlo, ci conviene considerarla come funzione parziale

$$\Gamma : \Sigma \to \Sigma_\perp \quad \sim \quad \Sigma \rightharpoonup \Sigma$$

In questo modo possiamo vedere $\varphi : \Sigma \rightharpoonup \Sigma$ come insieme di coppie $\langle\sigma, \sigma'\rangle$. Rappresentiamo la relazione con le regole di inferenza

$$\frac{\mathscr{B}\,[\![b]\!]\,\sigma \quad \sigma \xleftarrow{\mathscr{C}[\![c]\!]} \sigma'' \quad \sigma'' \to \sigma'}{\sigma \to \sigma'} \qquad \frac{\neg\mathscr{B}\,[\![b]\!]\,\sigma}{\sigma \to \sigma}$$

Se consideriamo $\mathscr{B}$ e $\mathscr{C}$ date, e costruiamo le regole su tali funzioni, otterremo un insieme di regole di inferenza infinito in cui ciascuna regola ha un numero finito di premesse.
Quindi per il teorema di Kleene possiamo concludere che $\hat{R}_\Gamma$, cioè l'insieme delle conseguenze immediate di R_Γ, è monotono e continuo, e che dunque R_Γ ha minimo punto fisso.

4.2. Equivalenza fra semantica operazionale e semantica denotazionale

Passiamo adesso a vedere come si dimostra l'equivalenza fra le due semantiche che abbiamo dato per IMP.
Il punto di partenza sono due cose fondamentalmente diverse fra di loro: da una parte abbiamo una manciata di formule del tipo $\langle e, x\rangle \to v$, che valutano una certa espressione e in uno stato σ e fanno corrispondere un valore v. Dall'altra abbiamo funzioni che vanno dal dominio sintattico di riferimento ad una funzione che prende lo stato e restituisce un risultato. I due oggetti sono sensibilmente differenti, e dimostrarne l'equivalenza ci porta ad una dimostrazione piuttosto insidiosa.
Vediamo come si dimostra nei tre domini.

4.2.1. Dimostrazione per $\mathscr{A}$ e $\mathscr{B}$

I due domini sintattici delle espressioni aritmetiche e booleane sono domini sui quali la nostra valutazione semantica va sempre a buon fine. Quindi possiamo limitarci a dimostrare la seguente proprietà.

$$P(a) \stackrel{\text{def}}{=} \langle a, \sigma \rangle \to \mathscr{A} [\![a]\!] \sigma \quad \forall a \in Aexpr$$

In pratica, vogliamo dimostrare che la valutazione dell'espressione aritmetica a nella memoria σ con la semantica denotazionale restituirebbe esattamente il valore restituito dall'applicazione della semantica denotazionale di a alla stessa memoria σ. Tornando alla metafora che vede la semantica operazionale come un'interprete e la semantica denotazionale come un compilatore, si controlla che l'esecuzione dell'interprete dia lo stesso risultato dell'esecuzione del programma compilato.

La prova è fatta banalmente per induzione strutturale. Vediamo alcune dimostrazioni.

Costanti Si dimostra facilmente mediante l'applicazione della semantica denotazionale che

$$P(n) \stackrel{\text{def}}{=} \langle n, \sigma \rangle \to \mathscr{A} [\![n]\!] \sigma = \langle n, \sigma \rangle \to n$$

Variabili Si dimostra facilmente mediante l'applicazione della semantica denotazionale che

$$P(x) \stackrel{\text{def}}{=} \langle x, \sigma \rangle \to \mathscr{A} [\![x]\!] \sigma = \langle x, \sigma \rangle \to \sigma x$$

Espressioni binarie In espressioni del tipo $a_0 \circ a_1$, dove $\circ$ è un generico operatore che corrisponde all'operatore vero $\odot$, per l'induzione strutturale possiamo assumere $P(a_0) = \langle a_0, \sigma \rangle \to n_0$ e $P(a_1) = \langle a_1, \sigma \rangle \to n_1$. Con queste premesse possiamo asserire che

$$P(a_0 \circ a_1) \stackrel{\text{def}}{=} \langle a_0 \circ a_1, \sigma \rangle \to \mathscr{A} [\![a_0 \circ a_1]\!] \sigma =$$
$$\langle a_0 \circ a_1, \sigma \rangle \to \mathscr{A} [\![a_0]\!] \sigma \odot \mathscr{A} [\![a_1]\!] \sigma = \langle a_0 \circ a_1, \sigma \rangle \to a_0 \odot a_1$$

Ovviamente il discorso sulle espressioni rimanenti è perfettamente analogo: tutte le espressioni binarie si dimostrano nella stessa maniera.

4.2.2. Dimostrazioni per $\mathscr{C}$

Il discorso per il dominio dei comandi è leggermente più complesso: un comando potrebbe non terminare. Fare una dimostrazione come quella precedente significherebbe solo che i due valori sono uguali, ma non assicurerebbe niente nel caso in cui un comando non termini. Noi vogliamo che

- se un comando c termina in semantica operazionale, termini in semantica denotazionale con lo stesso output
- se un comando c *non* termina in semantica operazionale, *non* termini neppure in semantica denotazionale.

Per farlo, occorre fare due dimostrazioni separate: dobbiamo dimostrare

- $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C} [\![c]\!] \sigma = \sigma'$, ovvero che il fatto che la semantica operazionale sul comando c con la memoria σ restituisca una memoria σ' implica che quest'ultimo debba essere risultato della semantica denotazionale applicata a σ.
- $P(c) \stackrel{\text{def}}{=} \mathscr{C} [\![c]\!] \sigma = \sigma' \Rightarrow \langle c, \sigma \rangle \to \sigma'$, ovvero il viceversa

Facendo questo siamo sicuri di aver dimostrato la doppia implicazione.

4.2.2.1. Dalla semantica operazionale alla denotazionale

Per questa prima dimostrazione, dovendo dimostrare proprietà di una regola, procediamo ovviamente per induzione sulle regole. Prenderemo dunque una regola, assumeremo valida la proprietà sulle premesse e dimostreremo la conseguenza in base a quanto assunto. Le dimostrazioni seguiranno questo schema.

- Prenderemo la regola della semantica operazionale che ha come conseguenza il costrutto sul quale lavoriamo
- Assumiamo vera la proprietà per le premesse
- Utilizziamo la semantica denotazionale dei costrutti, per dimostrare la proprietà.

Assegnamento Prendiamo la regola dell'assegnamento.

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma \left[{}^{m}/_{x} \right]}$$

Dobbiamo considerare già dimostrato (come fatto tra l'altro nella sezione scorsa) che $P(\langle a, \sigma \rangle \to m) \stackrel{\text{def}}{=} \mathscr{A} [\![a]\!] \sigma = m$. Vediamo brevemente che

$$P\left(\langle x := a, \sigma \rangle \to \sigma \left[{}^{m}/_{x} \right]\right) \stackrel{\text{def}}{=} \mathscr{C} [\![x := a]\!] \sigma = \sigma \left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x} \right] = \sigma \left[{}^{m}/_{x} \right]$$

che dimostra quel che volevamo dimostrare.

Concatenazione Prendiamo la regola della concatenazione

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Questa volta dobbiamo assumere che valgano $P\left(\langle c_0, \sigma \rangle \to \sigma''\right) \stackrel{\text{def}}{=} \mathscr{C} [\![c_0]\!] \sigma = \sigma''$ e l'analoga $P\left(\langle c_1, \sigma'' \rangle \to \sigma'\right)$. Vediamo che

$$P\left(\langle c_0; c_1, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C} [\![c_0; c_1]\!] \sigma = \mathscr{C} [\![c_1]\!]^* \left(\mathscr{C} [\![c_0]\!] \sigma\right) = \mathscr{C} [\![c_1]\!]^* \sigma'' = \sigma'$$

come volevasi dimostrare.

Condizionale Il condizionale è al solito diviso in due regole assolutamente speculari: la dimostrazione é la stessa per l'una o per l'altra regola. Dimostriamo la regola con la guardia vera.

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

Assumiamo la validità di $P(\langle b, \sigma \rangle \to \mathbf{true}) \stackrel{\text{def}}{=} \mathscr{B} [\![b]\!] \sigma = \mathbf{true}$ e $P\left(\langle c_0, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C} [\![c_0]\!] \sigma = \sigma'$. Vediamo che

$$P\left(\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C} [\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] \sigma = \mathscr{B} [\![b]\!] \to \mathscr{C} [\![c_0]\!] \sigma, \mathscr{C} [\![c_1]\!] \sigma = \mathscr{C} [\![c_0]\!] \sigma = \sigma'$$

come volevasi dimostrare.

Iterazione Il while va suddiviso nei suoi due casi, abbastanza differenti fra di loro, ma prima facciamo un ragionamento preliminare.

Sappiamo che $\mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = fix(\Gamma)$ con $\Gamma = \lambda\varphi.\lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$. Essendo la nostra semantica un punto fisso, deve valere la relazione

$$\mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] = \Gamma\mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!] =$$
$$\lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Notare che questa non è una definizione della semantica del while, bensì solamente una sua proprietà in quanto non definisce univocamente la semantica. Questa proprietà diventerà vera al momento dell'esecuzione, motivo per cui non può essere utilizzata per definire la semantica. Ma torniamo alla dimostrazione.

Partiamo dal caso in cui la guardia è falsa.

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma}$$

Al solito assumiamo $P(\langle b, \sigma\rangle \to \textbf{false}) \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma = \textbf{false}$, e dimostriamo $P(\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma)$.

La formula ricavata sopra diventa semplicemente

$$\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma = \sigma$$

in quanto $\mathscr{B}[\![b]\!]\sigma$ è falso per ipotesi.

Nel caso in cui la guardia sia vera abbiamo

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'}$$

Possiamo assumere la proprietà valida sulle premesse:

$$P(\langle b, \sigma\rangle \to \textbf{true}) = \mathscr{B}[\![b]\!]\sigma = \textbf{true}$$
$$P(\langle c, \sigma\rangle \to \sigma'') = \mathscr{C}[\![c]\!]\sigma = \sigma''$$
$$P(\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma') = \mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma'' = \sigma'$$

Riprendiamo la nostra riscrittura della regola del while.

$$\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma =$$
$$\mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma) =$$
$$\mathscr{C}[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma'' =$$
$$\sigma'$$

Notare che l'asterisco può essere tolto dal momento che siamo sicuri che il comando verrà eseguito su una memoria e non su $\perp$. La formula è dunque dimostrata anche in questo caso: possiamo essere sicuri che anche per il while le due semantiche sono esattamente equivalenti.

4.2.2.2. Dalla semantica denotazionale alla operazionale

Per dimostrare nel senso inverso, utilizziamo invece un'induzione strutturale. Per ogni produzione c dei comandi, dimostreremo

$$P(c) \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

cioè che se la semantica denotazionale di c restituisce σ' allora anche l'operazionale su c deve fare la stessa cosa.

Come prima, dimostriamolo per ogni categoria sintattica. La tecnica generale di dimostrazione sarà la seguente:

- assumeremo la premessa dell'implicazione, e la scomporremo in affermazioni più semplici
- vedremo che queste affermazioni più semplici corrispondono, per ipotesi induttiva, a formule ben formate della semantica operazionale
- utilizzeremo queste formule ben formate come premesse per dimostrare la conseguenza cercata

Skip Dobbiamo dimostrare

$$P(\mathbf{skip}) = \mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma' \Rightarrow \langle \mathbf{skip}, \sigma\rangle \to \sigma'$$

Per dimostrare l'implicazione, assumiamo vera l'ipotesi $\mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma$: viene immediato dimostrare che $\langle \mathbf{skip}, \sigma\rangle \to \sigma$ per definizione.

Abbiamo dunque facilmente dimostrato il fatto.

Assegnamento Dobbiamo dimostrare

$$P(x := a) = \mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma' \Rightarrow \langle x := a, \sigma\rangle \to \sigma'$$

Assumiamo ancora vera l'ipotesi, e dimostriamo la conseguenza. Per definizione, $\sigma' = \sigma[^n/_x]$ dove $\mathscr{A}\,[\![a]\!]\,\sigma = n$. Questo implica $\langle a, \sigma\rangle \to n$, che può essere utilizzato come premessa della regola $\langle x := a, \sigma\rangle \to \sigma[^n/_x]$ che è la formula che vogliamo dimostrare.

Concatenazione Dobbiamo dimostrare

$$P(c_1; c_2) = \mathscr{C}\,[\![c_1; c_2]\!]\,\sigma = \sigma' \Rightarrow \langle c_1; c_2, \sigma\rangle \to \sigma'$$

Al solito, assumiamo vera l'ipotesi. Sappiamo che $\mathscr{C}\,[\![c_1; c_2]\!]\,\sigma = \sigma' \quad \to \quad \mathscr{C}\,[\![c_2]\!]^*\,(\mathscr{C}\,[\![c_1]\!]\,\sigma) = \sigma'$. Visto che abbiamo supposto che la computazione termini, possiamo eliminare l'asterisco dalla seconda applicazione e prendere $\mathscr{C}\,[\![c_1]\!]\,\sigma = \sigma''$ e $\mathscr{C}\,[\![c_2]\!]\,\sigma'' = \sigma'$, che implicano rispettivamente $\langle c_1, \sigma\rangle \to \sigma''$ e $\langle c_2, \sigma''\rangle \to \sigma'$. Queste sono proprio le premesse della regola operazionale della concatenazione, che ci permettono di concludere $\langle c_1; c_2, \sigma\rangle \to \sigma'$, come volevasi dimostrare.

Condizionale Dobbiamo dimostrare

$$P(\mathbf{if}\, b\, \mathbf{do}\, c_1\, \mathbf{else}\, c_2) = \mathscr{C}\,[\![\mathbf{if}\, b\, \mathbf{do}\, c_1\, \mathbf{else}\, c_2]\!]\,\sigma = \sigma' \Rightarrow \langle \mathbf{if}\, b\, \mathbf{do}\, c_1\, \mathbf{else}\, c_2, \sigma\rangle \to \sigma'$$

Al solito, abbiamo due casi perfettamente speculari. Ci limitiamo a dare la dimostrazione per il caso in cui la guardia sia falsa.

Assumiamo la premessa. Vale dunque $\mathscr{C}\,[\![\mathbf{if}\, b\, \mathbf{do}\, c_1\, \mathbf{else}\, c_2]\!]\,\sigma = \sigma' = \mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c_1]\!]\,\sigma, \mathscr{C}\,[\![c_2]\!]\,\sigma = \mathscr{C}\,[\![c_2]\!]\,\sigma = \sigma'$, in quanto $\mathscr{B}\,[\![b]\!]\,\sigma = \mathbf{false}$. Però $\mathscr{B}\,[\![b]\!]\,\sigma = \mathbf{false} \Rightarrow \langle b, \sigma\rangle \to \mathbf{false}$ e $\mathscr{C}\,[\![c_2]\!]\,\sigma = \sigma' \Rightarrow \langle c_2, \sigma\rangle \to \sigma'$, che sono la premessa di $\langle \mathbf{if}\, b\, \mathbf{do}\, c_1\, \mathbf{else}\, c_2, \sigma\rangle \to \sigma'$ in semantica operazionale, come volevasi dimostrare.

Iterazione Dobbiamo dimostrare

$$P(\textbf{while}\, b\, \textbf{do}\, c) = \mathscr{C}\, [\![\textbf{while}\, b\, \textbf{do}\, c]\!]\, \sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma'$$

Vediamo di trasformarla in una forma più comoda. Il problema è che nella forma classica, che utilizza il punto fisso, non sono esplicitate le iterazioni, che al contrario sono esplicitate nella semantica operazionale. Abbiamo che

$$\begin{aligned} (\mathscr{C}\, [\![\textbf{while}\, b\, \textbf{do}\, c]\!]\, \sigma = \sigma' \quad &\Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma') &\Leftrightarrow \\ (fix\, \Gamma\, \sigma = \sigma' \quad &\Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma') &\Leftrightarrow \\ \left(\left(\bigsqcup_{i\in\omega} \Gamma^n\right) \sigma = \sigma' \quad \right. &\Rightarrow \quad \left. \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma' \right) &\Leftrightarrow \\ (\forall n. \Gamma^n \sigma = \sigma' \quad &\Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma') \end{aligned}$$

Su quest'ultima definizione possiamo applicare un'ulteriore induzione matematica *sopra* l'induzione strutturale che stiamo già eseguendo. L'induzione matematica ci assicurerà la validità della proprietà P per tutte le iterazioni precedenti del while, l'induzione strutturale si occuperà come sempre di assicurare tale validitá per il resto.

Procediamo con l'induzione matematica.

P(0) Banalmente abbiamo

$$\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma'$$

che è sempre vera in quanto la parte destra dell'implicazione é sempre falsa ($\forall x. \bot x = \bot$)

P(n) ⇒ P(n+1) Assumiamo come vera $A(n) \stackrel{\text{def}}{=} \Gamma^n \bot\sigma = \sigma' \quad \Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma'$, e dimostriamo $A(n+1) \stackrel{\text{def}}{=} \Gamma^{n+1} \bot\sigma = \sigma' \quad \Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma'$. Al solito, assumiamo la premessa: questa può essere riscritta come $\Gamma\, (\Gamma^n \bot)\, \sigma = \sigma'$, ovvero

$$\mathscr{B}\, [\![b]\!]\, \sigma \rightarrow (\Gamma^n \bot)^* \, (\mathscr{C}\, [\![c]\!]\, \sigma)\, , \sigma = \sigma'$$

Vediamo adesso i due casi in cui b corrisponde a **true** o **false**.

- Se $\mathscr{B}\, [\![b]\!]\, \sigma = \textbf{false}$, abbiamo $\sigma' = \sigma$. Ma possiamo assumere per l'ipotesi dell'induzione strutturale che $\langle b, \sigma\rangle \rightarrow \textbf{false}$, che è proprio la premessa della regola $\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \rightarrow \sigma$: é esattamente quel che volevamo dimostrare!
- Se $\mathscr{B}\, [\![b]\!]\, \sigma = \textbf{true}$, dobbiamo rieseguire il while: abbiamo $\sigma' = (\Gamma^n \bot)^* \, (\mathscr{C}\, [\![c]\!]\, \sigma)$. Al solito, possiamo togliere l'asterisco in quanto siamo sicuri che tutto converge ad un σ' e non c'è bisogno di considerare il caso $\bot$. Possiamo dunque dire che $\mathscr{C}\, [\![c]\!]\, \sigma = \sigma''$ e dedurre, grazie all'ipotesi dell'induzione *strutturale*, che $\langle c, \sigma\rangle \rightarrow \sigma''$. Inoltre, possiamo sempre per l'induzione *strutturale* dedurre che $\langle b, \sigma\rangle \rightarrow \textbf{true}$. A questo punto, interviene l'induzione matematica: sappiamo che vale $A(n) = \Gamma^n \bot\sigma'' = \sigma' \quad \Rightarrow \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \rightarrow \sigma'$. Abbiamo dedotto dunque le tre premesse della regola del while vero, e possiamo considerarne vera la conseguenza.

4.3. Dimostrare proprietà con la semantica denotazionale

Per dimostrare le proprietà della semantica denotazionale abbiamo bisogno in generale di poter provare queste proprietà per i punti fissi, e in particolare vogliamo poter studiare i *funzionali* attraverso i quali arriviamo a questi punti fissi.

Definizione 4.2 (Induzione computazionale)
L'induzione computazionale afferma che

$$\frac{P \text{ inclusivo} \quad \bot \in P \quad \forall d \in D.(d \in P \Longrightarrow F(d) \in P)}{fix(F) \in P}$$

Dimostrazione. Dalle ipotesi abbiamo che $\bot \in P$ e inoltre $F(\bot) \in P$ da cui per induzione matematica $\forall n \in \omega\ F^n(\bot) \in P$. Le approssimazioni di F, come noto, formano una catena, il cui limite $\bigsqcup_{x \in \omega} F^n(\bot) = fix(F) \in P$ per l'inclusività (3.22) di P. ☠

Esempio 4.3 (Esempio)
[Applicazione induzione computazionale 1]

$$\mathscr{C}\,[\![\mathbf{while}\,b\,\mathbf{do}\,c]\!]\,\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \to \sigma'$$

Dobbiamo dimostrare che

$$(\bigsqcup \Gamma^n \bot)\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \to \sigma'$$

con

$$\Gamma\varphi\sigma = \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Definiamo allora questa proprietà del funzionale

$$P(\varphi) \stackrel{\text{def}}{=} \forall\sigma.(\varphi\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \to \sigma')$$

Cioè all'approssimazione $\sigma\sigma'$ anche l'operazionale va da $\sigma\sigma'$. Se il limite $\bigsqcup \Gamma^n \bot$ appartiene ancora a questa proprietà, la dimostrazione è fatta.
Per farlo vogliamo usare l'induzione computazionale e quindi dobbiamo avere

- *un CPO, che è il $\Sigma \to \Sigma_\bot$, dove sono definite le φ.*
- *una funzione monotona continua*

$$\Gamma : (\Sigma \to \Sigma_\bot) \to (\Sigma \to \Sigma_\bot)$$

- *l'insieme P in cui vale la nostra proprietà deve essere inclusivo*

Mostriamo l'ultimo punto, l'inclusività di P

$$\forall i \quad \varphi_i\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \to \sigma') \stackrel{?}{\Longrightarrow} (\bigsqcup \varphi_i)\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \to \sigma'$$

Assumiamo le premesse globali, e assumiamo la premessa della seconda parte. Rimane da dimostrare

$$\langle \mathbf{while}\,b\,\mathbf{do}\,c, \sigma\rangle \stackrel{?}{\to} \sigma'$$

Nel nostro ordimanento, se $(\bigsqcup \varphi)\sigma = \sigma'$, allora esiste un particolare k tale che, $\varphi_k\sigma = \sigma'$, cioè la proprietà vale da quel k in poi, ma abbiamo assunto che

$$\forall i \quad \varphi_i\sigma = \sigma' \Longrightarrow \langle \mathbf{while}\,b\,\mathbf{do}\,c\sigma\rangle \to \sigma'$$

e quindi P è inclusivo.

Possiamo quindi possiamo utilizzare finalmente l'induzione computazionale. Dimostriamo che tutte le premessa sono verificate

- $\bot \in P$
  cioè $P(\bot)$
  Deve essere
  $$\bot\sigma = \sigma' \overset{?}{\Longrightarrow} \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$$
  Ma $\bot\sigma$ *non può essere uguale a* σ'*, quindi la proprietà è rispettata.*
- $\forall d \in D.d \in P \Longrightarrow F(d) \in P$
  cioè $P(\varphi) \overset{?}{\Longrightarrow} P(\Gamma\varphi)$
  Riscriviamola ancora come
  $$\forall \sigma, \sigma' \quad \mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}\,[\![c]\!]\,\sigma), \sigma = \sigma' \overset{?}{\Longrightarrow} \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$$
  Assumiamo la premessa
  $$P(\varphi) = \varphi\sigma = \sigma' \Longrightarrow \langle \textbf{while}\, b\, \textbf{do}\, c\rangle, \to \sigma'$$
  Caso false
  $$\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false} \quad \sigma = \sigma'$$
  Caso true
  $$\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{true} \quad \varphi^*(\mathscr{C}\,[\![c]\!]\,\sigma) = \sigma'$$
  La stella si può buttare.
  $$\varphi^*\,\underbrace{(\mathscr{C}[\![c]\!]\sigma)}_{\sigma''} = \sigma'$$
  quelle che resta è
  $$\varphi\sigma'' = \sigma'$$
  e abbiamo
  $$\langle c, \sigma\rangle \to \sigma''$$
  $$\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma'' s\rangle \to \sigma'$$
  $$\mathscr{B}[\![b]\!] = \textbf{true}$$
  allora
  $$\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$$

====PEER REVIEW=========

Esempio 4.4 (Applicazione induzione computazionale 2)

$$\mathscr{C}\,[\![\textbf{while}\, x \neq 0\, \textbf{do}\, x := x - 1]\!]\,\sigma = \sigma' \Longrightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$

Abbiamo che:

$$\mathscr{C}\,[\![w]\!] = fix(\Gamma) \quad \Gamma\varphi\sigma = \sigma x \neq 0 \to \varphi\sigma[^{\sigma x - 1}/_x], \sigma$$

Sappiamo che $\Sigma \to \Sigma_\bot$ *è un CPO e che* Γ *è monotona, continua. Riscriviamo la proprietà:*

$$P(\varphi) \overset{\text{def}}{=} \forall \sigma.(\varphi\sigma = \sigma' \Longrightarrow \sigma x \geq \wedge \sigma' = \sigma[^0/_x])$$

Dobbiamo dimostrare l'inclusività di φ

$$\Big(\forall i \forall \sigma. \big(\varphi_i \sigma = \sigma' \Longrightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\big)\Big) \Longrightarrow$$
$$\Longrightarrow \forall \sigma. \Big((\textstyle\bigsqcup_{i \in \omega} \varphi_i)\sigma = \sigma' \overset{\sigma}{\Longrightarrow} x \geq 0 \wedge \sigma' = \sigma[^0/_x]\Big)$$

Assumiamo le due premesse e dimostriamo $\sigma x \geq 0$ *e* $\sigma' = \sigma[^0/_x]$. *Abbiamo che*

$$\forall i. \forall \sigma \quad \varphi_i \sigma = \sigma' \Longrightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x] \qquad \bigsqcup \varphi_i \sigma = \sigma'$$

Allora visto che vale al limite, vale anche a partire da un certo k

$$\exists k. \varphi_k \sigma = \sigma'$$

e quindi le due implicazioni valgono da quel k fino al limite.

Possiamo utilizzare l'induzione computazionale

- $P(\bot)$ *ovvio,* $\bot\sigma = \sigma'$
- $P(\varphi) \stackrel{?}{\Longrightarrow} P(\Gamma\varphi)$

Assumiamo

$$\forall \sigma. (\varphi\sigma = \sigma' \Longrightarrow \sigma x \geq 0 \wedge \sigma; = \sigma[^0/_x]) = P(\varphi)$$

Dimostriamo

$$\Gamma\varphi\sigma = \sigma' \Longrightarrow \sigma x \geq_? 0 \wedge \sigma' \stackrel{?}{=} \sigma[^0/_x]$$

Assumiamo

$$\sigma x \neq 0 \rightarrow \varphi\sigma[^{\sigma x - 1}/_x], \sigma = \sigma'$$

Primo caso*:* $\sigma x = 0$

$$\sigma = \sigma' \quad \sigma x \geq 0 \quad \sigma = \sigma[^0/_x]$$

visto che $\sigma x = 0$ *allora* $\sigma x \geq 0$

Secondo caso*:* $\sigma x \neq 0$

$$\varphi \underbrace{\sigma[^{\sigma x - 1}/_x]}_{\sigma''} = \sigma'$$

ho assunto $P(\varphi)$ $\forall\sigma$*, assumiamola in particolare per* σ''
Abbiamo allora

$$\sigma[^{\sigma x - 1}/_x]x \geq 0 \quad \sigma x - 1 \geq 0 \quad \sigma x \geq 1$$

$$\sigma' = \sigma[^{\sigma x - 1}/_x][^0/_x] = \sigma[^0/_x]$$

Visto che P vale anche per il limite allora vale anche per la semantica del while.

Parte II.

Il linguaggio HOFL

5. Sintassi e semantica operazionale di HOFL

5.1. λ-calcolo

Introduciamo il λ-calcolo non tipato, un formalismo introdotto da Alonzo Church negli anni ’30 come linguaggio formale per la definizione e applicazione di funzioni con ricorsione. Il λ-calcolo possiede un’espressività equivalente a quella delle macchine di Turing.

5.1.1. λ-calcolo non tipato

Diamo per prima cosa la grammatica del termine t, unica categoria sintattica del linguaggio.

$$t ::= x \quad | \quad \lambda x.t \quad | \quad t_1\ t_2 \quad | \quad c \quad | \quad t_0 \to t_1, t_2$$

In questa definizione viene descritto il termine come unione di

- variabili x
- *astrazione* di una funione t con parametro formale x
- *applicazione* della funzione restituita da un termine t_1 su un argomento restituito dal termine t_2
- costante c
- comando condizionale che valuta t_0 e
  - se vero esegue t_1
  - se falso esegue t_2

Diamo alcune definizioni utili.

Definizione 5.1 (Variabili libere)
Definiamo insieme delle variabili libere in un termine l’insieme così costituito:

- $f_v(x) = \{x\}$
- $f_v(\lambda x.t) = f_v(t) \setminus \{x\}$
- $f_v(t_1\ t_2) = f_v(t_1) \cup f_v(t_2)$

L’insieme delle variabili libere è l’insieme delle variabili che compaiono all’interno di un’espressione senza essere legate da un λ. Passiamo all’operazione di sostituzione.

Definizione 5.2 (Sostituzione)
L’operazione di sostituzione è così definita:

costante $c\,[^t/_x] = c \quad se\ c \neq x$

variabile $x\,[^t/_x] = t$

applicazione $(t_1 t_2)\,[^t/_x] = t_1\,[^t/_x]\,t_2\,[^t/_x]$

astrazione $(\lambda x.t)\left[^{t_1}/_y\right] = \lambda x.t\left[^{t_1}/_y\right]$ *se e solo se* $y \neq x$ *e* $x \notin f_v(t_1)$

Per non incappare nelle condizioni della quarta regola, possiamo utilizzare l'operazione di α-conversione, che permette sostanzialmente di cambiare il nome di una variabile. Vediamola formalmente.

Definizione 5.3 (α-conversione)
$\lambda x.t = \lambda y.t\,[^y/_x]$ *se* $y \notin f_v(t)$

Se y non è una variabile libera all'interno di un termine t possiamo sostituire y a tutte le occorrenze di x. In questo caso y viene detta una variabile *fresh* e partiamo dal presupposto che esistano infinite variabili e quindi è sempre possibile sceglierne una fresh.

Proprio per la natura intuitiva di questa operazione, in generale due termini vengono considerati uguali a meno di α-conversione.

L'operazione di β-conversione è invece un'operazione che corrisponde al passaggio di parametri: all'applicazione di una funzione su un argomento, sostituisce all'interno del corpo della funzione tutte le occorrenze del parametro formale con il valore dell'argomento.

Definizione 5.4 (β-conversione (copy-rule o β-reduction))
$(\lambda x.t_1)\,t_2 = t_1[^{t_2}/_x]$

5.1.2. λ-calcolo semplicemente tipato

Il λ-calcolo semplicemente tipato è un'estensione del λ-calcolo non tipato, visto precedentemente, a cui viene aggiunto un semplice sistema di tipi: partendo da una base di tipi B (nel caso di HOFL $B = \{int\}$) e dal tipo funzione $\rightarrow$ ogni termine ha un tipo

$$\tau = \tau \rightarrow \tau \mid T \quad \text{con} T \in B$$

Questo tipaggio viene detto semplice perché ogni termine ha esattamente *un* tipo mentre esistono tipaggi che prevedono tipi polimorfi o tipi dipendenti.

Il λ-calcolo semplicemente tipato ha la proprietà di essere *strongly normalizing* e cioè termini ben tipati si riducono sempre ad un valore (o forma normale). Questo si verifica perché la ricorsione non è ammessa dal sistema di tipi. In particolare non è possibile tipare

- il *looping term*
$$I = \lambda x.\ x\ x$$
- l'operatore di punto fisso, una funzione che calcola il punto fisso di un altra funzione.

Questo implica che un calcolo termina sempre e quindi il formalismo non è Turing equivalente.

Esempio 5.5 (looping term)
Il seguente termine viene detto looping term *perché se applicato a se stesso cicla infinitamente:*

$$I = \lambda x. \quad x \quad x$$

$$II = (\lambda x. \quad x \quad x)(\lambda x. \quad x \quad x) = II$$

per questo termine la semplificazione per β-conversione non termina.
Nel λ-calcolo semplicemente tipato la tipizzazione di questo termine fallisce con occur-check*, per tiparlo correttamente infatti è necessario che un termine acquisti due tipi differenti contemporaneamente, cosa possibile solo in un sistema di tipi polimorfo.*

Questo è un esempio artificioso per mostrare che il λ-calcolo non tipato permette computazioni infinite. Vediamone uno più significativo.

Esempio 5.6 (Y combinator)
Un fix point combinator *o Y combinator è una funzione di ordine superiore che può calcolare il punto fisso di una funzione, vediamo quello proposto da Haskell Curry per il λ-calcolo non tipato:*

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$$

Questo termine non è tipabile nel λ-calcolo semplicemente tipato per le stessi ragioni del precedente.

5.2. HOFL

HOFL (High Order Functional Language) è un linguaggio funzionale di ordine superiore, lazy, call-by-name, ed è sostanzialmente una variante del λ-calcolo semplicemente tipato, con alcune operazioni sugli interi e la possibilità di definire funzioni ricorsive.

In IMP si potevano interpretare come tipi le 3 categorie sintattiche: $\mathscr{A}exp, \mathscr{B}exp, \mathscr{C}om$. In HOFL al contrario abbiamo una sola categoria sintattica, il *termine*, che può essere tipato in diversi modi: abbiamo dei *costruttori di tipo* che definiscono un dominio come:

- **Prodotto Cartesiano** di 2 domini $D \times E$
- **Funzioni** tra 2 domini $D \to E$

Come vedremo nella sezione 6.1, questo porterà alla necessità di estendere la nostra teoria dei domini in modo da farle costruire un dominio adatto al nostro tipo qualunque esso sia, e questo è dovuto al fatto che possiamo generare infiniti tipi differenti.

5.2.1. Sintassi

Vediamo la sintassi del nostro linguaggio HOFL. Abbiamo due categorie: quella dei *tipi* e quella del linguaggio. La categoria dei tipi indica come costruire un qualunque tipo.

$$\tau := \quad int \quad | \quad \tau_1 \times \tau_2 \quad | \quad \tau_1 \to \tau_2$$

Abbiamo che un tipo può essere un intero, un prodotto di due tipi o una funzione da tipo a tipo. Esempi di tipo validi possono essere $int- > int$, $(int \times int) \to (int \times int)$, $int \to ((int \times int) \to int)$ e via dicendo.

Passiamo invece al linguaggio vero e proprio. Diamo qua una cosiddetta pre-sintassi: può definire anche strutture non valide in quanto contenenti errori di tipo. Vedremo a breve cosa questo significhi.

$t ::$	$x \quad \mid$	variabile
	$n \quad \mid$	costante
	$t_1 + t_2 \quad \mid \quad t_1 - t_2 \quad \mid \quad t_1 \times t_2 \quad \mid$	operazioni aritmetiche
	$\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \quad \mid$	condizionale
	$(t_1, t_2) \quad \mid \quad \mathbf{fst}(t) \quad \mid \quad \mathbf{snd}(t) \quad \mid$	operazioni su coppie
	$\lambda x.t \quad \mid \quad (t_1\ t_2) \quad \mid$	definizione e applicazione di funzione
	$\mathbf{rec}\ x.t$	funzioni ricorsive

Nel costrutto **if** il test ha successo con 0.
In HOFL τ è una *variabile di tipo* che corrisponde ad uno e un solo tipo ben preciso, mentre in altri linguaggi è possibile utilizzare tipi polimorfi.

5.2.2. Regole di inferenza di tipi

Le regole di inferenza dei tipi servono a verificare che un termine sintatticamente corretto, abbia effettivamente senso, e permettono quindi di evitare espressioni erronee come $3 + \lambda x.\ 3$ che infatti risulta non tipabile.
Le variabili x del linguaggio sono tipate e il tipo è dato dalla funzione *type*.

Variabili $x : type(x) = \hat{x}$

Operazioni

$$n : int \qquad \frac{t_1 : int \quad t_2 : int}{t_1 \text{ op } t_2 : int} \qquad \frac{t_0 : int \quad t_1 : \tau \quad t_2 : \tau}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : \tau} \qquad \text{con op} = +, -, \text{x}$$

Coppie

$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1, t_2) : (\tau_1 * \tau_2)} \qquad \frac{t : \tau_1 * \tau_2}{\textbf{fst}(t) : \tau_1} \qquad \frac{t : \tau_1 * \tau_2}{\textbf{snd}(t) : \tau_2}$$

Funzioni

$$\frac{x : \tau_1 \quad t : \tau_2}{\lambda x.t : \tau_1 \to \tau_2} \qquad \frac{t_1 : \tau_1 \to \tau_2 \qquad t_2 : \tau_1}{(t_1\ t_2) : \tau_2}$$

Rec

$$\frac{x : \tau \quad t : \tau}{\textbf{rec } x.t : \tau}$$

5.2.3. Problema della tipizzazione

5.2.3.1. Tipizzazione alla Church

Per ogni variabile è necessario indicare il corrispondente tipo.
I tipi vengono dedotti per ricorsione strutturale, perché dato il tipo dei sottotermini, esiste una e una sola regola per derivare il tipo di un termine.

Vediamo ad esempio il fattoriale:

```
fact(x) = if x=0 then 1 else x X fact(x-1)
```

Esempio 5.7 (Programma tipabile (Church))
Sia x intero, f funzione da intero ad intero. Allora deduciamo i tipi

$$\textbf{fact} \stackrel{\text{def}}{=} \textbf{rec} \underbrace{f}_{int \to int} . \underbrace{\lambda \underbrace{x}_{int} . \textbf{if } \underbrace{x}_{int} \textbf{ then } 1 \textbf{ else } \underbrace{x}_{int} \times (\underbrace{\underbrace{f}_{int \to int} \ \underbrace{(\underbrace{x}_{int} - \underbrace{1}_{int})}_{int}}_{int})}_{int \to int} \quad : int \to int$$

5.2.3.2. Tipizzazione alla Curry

Viene utilizzata nei linguaggi funzionali. Inferisce il tipo di termini non tipati, il programmatore in generale è sollevato dal problema ma può intervenire. $x : \tau$: può essere di qualunque tipo
(x y) invece non so il tipo preciso ma so che

$$(\underbrace{x}_{\tau_1 \to \tau_2} \underbrace{y}_{\tau_1}) : \tau_2$$

Esempio 5.8 (Programma non tipabile (Curry))

$$\textbf{fact}(x) \stackrel{\text{def}}{=} f(f,x) \quad \textbf{where} \quad f(g,x) \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \times g(g, x-1)$$

Questa definizione del fattoriale non è ricorsiva!
Può essere scritta in un linguaggio come Algol, mentre non è tipabile in HOFL:
Riscriviamola in HOFL

$$\textbf{fact} \stackrel{\text{def}}{=} \lambda x.(f(f,x))$$

$$f \stackrel{\text{def}}{=} \lambda y.\, \textbf{if } \textbf{snd}(y) \textbf{ then } 1 \textbf{ else } \textbf{snd}(y) \times \underbrace{(\underbrace{\textbf{fst}(y)}_{t_1:\tau_1} \underbrace{(\textbf{fst}(y), \underbrace{\underbrace{\textbf{snd}(\underbrace{y}_{t_0:\tau_0})}_{t_2:\tau_2} - \underbrace{1}_{t_3:\tau_3}}_{t_4:\tau_4})}_{t_5:\tau_5})}_{t_6:\tau_6}$$

Abbiamo tipato solo la parte finale perché è quella problematica.

$-1 : \tau_3$	$\tau_3 : int$
$y : \tau_0$	
$\textbf{fst}(y) : \tau_1$	$\tau_0 = \tau_1 * \tau_7$
$\textbf{snd}(y) : \tau_2$	$\tau_0 = \tau_8 * \tau_2$
$\textbf{snd}(y) - 1 : \tau_4$	$\tau_4 = int; \tau_2 = int; \tau_3 = int$
$(\textbf{fst}(y), \textbf{snd}(y) - 1) : \tau_5$	$\tau_5 = \tau_1 * \tau_4$
$(\textbf{fst}(y)(\textbf{fst}(y), \textbf{snd}(y) - 1)) : \tau_6$	$\tau_1 = \tau_5 \to \tau_6$

Ricaviamo un sistema di equazioni

$$\begin{cases} \tau_0 = \tau_1 * \tau_7 \\ \tau_0 = \tau_8 * \tau_2 \\ \tau_2 = int \\ \tau_3 = int \\ \tau_4 = int \\ \tau_5 = \tau_1 * \tau_4 \\ \tau_1 = \tau_5 \to \tau_6 \end{cases}$$

Problema di unificazione.

1) *Se ho $x = x$ con x varaibile, è sempre vera e la posso cancellare.*

2) $f(t_1, \ldots, t_n) = f'(t'_1, \ldots, t'_m)$
Se $f \neq f'$, non è mai possibile unificarle, quindi posso non guardare i termini t_i. Se $f = f'$ allora $m = n$ perché un costruttore ha una sola arità, e allora bisogna avere

$$\tau_1 = \tau'_1, \tau_2 = \tau'_2, \ldots, \tau_n = \tau'_n$$

3) $x = t$ si sostituisce ogni occorrenza di x con t.
Cosa succede se $x = t(x)$?
Non è possibile unificare.
Perché quando sostituisco invece di eliminare la variabile, la ritrovo in un termine più complesso, e questo processo non ha soluzione.

Riassumendo nell' algoritmo di unificazione

1) *eliminare $x = x$*
2) *ridurre $f(x) = f(y) \Rightarrow x = y$*
3) *sostituire $x = f(y) \quad x \neq FV(y)$ in tutte le altre occorrenze*

L'ordine col quale si risolvono le equazioni non cambia il risultato, ma può rendere il calcolo lineare con la dimensione del sistema.

$$\begin{cases} \tau_0 = \tau_1 * \tau_7 \\ \tau_0 = \tau_8 * \tau_2 \\ \tau_2 = int \\ \tau_3 = int \\ \tau_4 = int \\ \tau_5 = \tau_1 * \tau_4 \\ \tau_1 = \tau_5 \to \tau_6 \end{cases} \Rightarrow \begin{cases} \tau_0 = \tau_1 * \tau_7 \\ \tau_0 = \tau_8 * int \\ \tau_1 = \tau_5 \to \tau_6 \\ \tau_5 = \tau_1 * int \end{cases} \Rightarrow \begin{cases} \tau_0 = \tau_1 * \tau_7 \\ \tau_8 * int = \tau_1 * \tau_7 \\ \tau_1 = \tau_5 \to \tau_6 \\ \tau_5 = \tau_1 * int \end{cases} \Rightarrow$$

$$\begin{cases} \tau_0 = \tau_1 * \tau_7 \\ \tau_8 = \tau_1 \\ \tau_7 = int \\ \tau_1 = \tau_5 \to \tau_6 \\ \tau_5 = \tau_1 * int \end{cases} \Rightarrow \begin{cases} \tau_0 = \tau_1 * int \\ \tau_1 = \tau_5 \to \tau_6 \\ \tau_5 = \tau_1 * int \end{cases} \Rightarrow \begin{cases} \tau_0 = \tau_1 * int \\ \tau_1 = (\tau_1 * int) \to \tau_6 \\ \tau_5 = \tau_1 * int \end{cases}$$

Ma

$$\tau_1 = (\tau_1 * int) \to \tau_6$$

non può essere risolta, in quanto si ricade nel 3o caso (self-dependency): τ_1 è argomento di se stessa. L'altezza dell'albero generato è infinita.

Esempio 5.9 (Programma non tipabile)
Un altro esempio interessante di termine non tipabile, è la lista infinita dei numeri pari $(t, 0)$ con

$$t = \mathbf{rec}\, P.\lambda x.(x(P(x+2))) = t$$

In realtà si tratta di una lista di liste

$$(0, (2, (4, (6 \dots$$

TODO scrivere bene passaggi da appunti montanari.

5.2.4. Sostituzione

Definiamo la sostituzione per ricorsione strutturale su tutti i costrutti.

$$
\begin{array}{ll}
x[^t/_x] = t & \\
y[^t/_x] = y & x \neq y \\
n[^t/_x] = n & \\
(t_1 + t_2)[^t/_x] = t_1[^t/_x] + [^t/_x] & \text{idem per } -, \times, \textbf{if then else}, \textbf{fst}, \textbf{snd} \\
(\lambda x.t)[^{t'}/_y] = \begin{cases} (x \cdots t) & x \text{ ricavabile} \\ \lambda z.t[^z/_x][^{t'}/_y] & z \notin FV(\lambda x.t), FV(t') \end{cases} & \text{si usa la prima con tale proprietà}
\end{array}
$$

TODO: sistemare il rec, non si capisce che intende con ricavabile.

Gli unici casi complessi sono λ e *rec* (*binders*), dove bisogna prima fare α-conversione con una variabile z *fresh* e poi applicare la sostituzione.

Come interagiscono sostituzione e typing?

Teorema 5.10 (La sostituzione rispetta i tipi)

$$t : \tau \wedge x, y : \tau' \implies t[^x/_y] : \tau$$

Il tipo del termine non cambia: (si dimostra per ricorsione strutturale).

TODO: dimostrazione è sulle dispense.

Alla luce di questo teorema possiamo ora definire la regola di α conversione.

Definizione 5.11 (α-conversione tipata)

$$\lambda x.t \equiv \lambda y.t[^y/_x] \quad se\ y \notin FV(t)\ \wedge\ type(x)=type(y)$$

5.3. Semantica operazionale di HOFL

Andiamo ora a definire la semantica operazionale di HOFL. Prima cosa da notare è che la nostra semantica operazionale tratta solamente termini chiusi, cioè termini t in cui $FV(t) = \emptyset$. Infatti non è presente una regola per le variabili.
Questo significa che la semantica operazionale è completamente indipendente dall'ambiente ρ.

Le formule ben formate della nostra semantica operazionale saranno della forma $t \to c$, ed indicheranno che il termine t si riduce alla forma canonica c.

Definiamo gli insiemi delle variabili libere e delle variabili legate in HOFL.

Definizione 5.12 (Variabili libere in un termine HOFL)
L'insieme delle variabili libere in un termine in HOFL è così ricorsivamente definito:

$$
\begin{array}{l}
FV(n) = \emptyset \\
FV(x) = \{x\} \\
FV(t_1 \,\textbf{op}\, t_2) = FV(t_1) \cup FV(t_2) \\
FV(\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) = FV(t_0) \cup FV(t_1) \cup FV(t_2) \\
FV(\textbf{fst}(t)) = FV(\textbf{snd}(t)) = FV(t) \\
FV(\lambda x.t) = FV(t) \backslash \{x\} \\
FV((t_1\ t_2)) = FV(t_1) \cup FV(t_2) \\
FV(\textbf{rec}\ x.t) = FV(t) \backslash \{x\}
\end{array}
$$

Definizione 5.13 (Variabili legate in un termine HOFL)
L'insieme delle variabili legate in un termine HOFL è così ricorsivamente definito:

$$
\begin{aligned}
&BV(n) = BV(x) = \emptyset \\
&BV(t_1 \mathbf{~op~} t_2) = BV(t_1) \cup BV(t_2) \\
&BV(\mathbf{if~} t_0 \mathbf{~then~} t_1 \mathbf{~else~} t_2) = BV(t_1) \cup BV(t_2) \cup BV(t_3) \\
&BV(\mathbf{fst}(t)) = BV(\mathbf{snd}(t)) = BV(t) \\
&BV(\lambda x.t) = BV(t) \cup \{x\} \\
&BV((t_1\ \ t_2)) = BV(t_1) \cup BV(t_2) \\
&BV(\mathbf{rec~} x.t) = BV(t) \cup \{x\}
\end{aligned}
$$

5.3.1. Forme Canoniche

Abbiamo detto che vogliamo ridurre i nostri termini a forme canoniche c. Ma quali sono queste forme canoniche? Vediamole.

Definizione 5.14 (Forme canoniche)
Le forme canoniche sono i termini che appartengono all'insieme C. L'appartenenza a tale insieme è definita dalle seguenti regole di inferenza:

$$\frac{}{n \in C_{int}}$$

$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1, t_2 : \ closed}{(t_1, t_2) \in C_{\tau_1 * \tau_2}}$$

$$\frac{\lambda x.t : \tau_1 \to \tau_2 \quad \lambda x.t \ closed}{\lambda x.t \in C_{\tau_1 \to \tau_2}}$$

Si tratta dunque di tutti i termini che sono

- interi
- coppie di termini chiusi
- funzioni chiuse

Notare che la scelta delle coppie come forme canoniche è puramente arbitraria, potremmo entrare nelle coppie e cercare di procedere ulteriormente. Al contrario per quanto riguarda le funzioni non abbiamo scelta, infatti cercare di portare una funzione in forma canonica, significa mostrare che due programmi differenti calcolano la stessa funzione. Noi usiamo una ricorsione infinita numerabile mentre i programmi non lo sono quindi fare la prova è impossibile! TODO: verificare ed elaborare

5.3.2. Regole di inferenza

Vediamo adesso le regole di inferenza di HOFL.

La prima regola è un assioma: afferma che una qualunque forma canonica si riduce a se stessa.

$$\frac{}{c \to c}$$

è una regola abbastanza ovvia. Passando a regole un po' più complesse, vediamo le tre regole delle operazioni.

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \underline{op} n_2} \qquad \frac{t_0 \to 0 \quad t_1 \to c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_1} \qquad \frac{t_0 \to n \quad n \neq 0 \quad t_2 \to c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_2}$$

La prima regola tutte le operazioni binarie, ed è praticamente analoga a quella definita per IMP. Anche il comando condizionale è analogo, tranne per il fatto che qua non esistono valori booleani e viene considerato 0 come true e qualunque altro valore come false.

Le regole che governano le coppie di elementi sono le seguenti.

$$\frac{t \to (t_1, t_2) \quad t_2 \to c_2}{\textbf{fst}(t) \to c_1} \qquad \frac{t \to (t_1, t_2) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_2}$$

La coppia semplice di elementi è forma canonica per assioma. Esistono anche le due operazioni che "estraggono" un elemento dalla coppia: le operazioni di first e second. La loro semantica è qua sopra.

Passiamo alla valutazione di funzione. Esistono due varianti di questa regola: vediamole ed analizziamone le differenze.
La prima regola è detta *lazy* ed è quella che utilizzeremo sempre:

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[{}^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

Questa regola sostituisce l'*intero* termine t_2 all'interno di t_1'. Vediamo la regola cosiddetta *eager*

$$\frac{t_1 \to \lambda x.t_1' \quad t_2 \to c_2 \quad t_1'[{}^{c_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

Questa regola prima porta t_2 in forma canonica e poi lo sostituisce all'interno di t_1'. Capiremo bene la differenza fra le due regole nell'esempio 5.16, dove vedremo che

- la valutazione lazy è generalmente più macchinosa della valutazione eager
- la valutazione eager porta a non calcolare più volte la stessa derivazione ma a volte può non terminare.

Infine per la regola ricorsiva facciamo un calcolo di punto fisso, il quale, se esiste, è la forma canonica. Va notato tuttavia che questo non è in generale il minimo punto fisso, mentre vedremo che lo sarà nella semantica denotazionale.
TODO: elaborare
Il funzionamento è semplicissimo: prende il corpo della funzione ricorsiva e lo sostituisce al posto della variabile sulla quale ricorriamo, in pratica facciamo l'*unfolding* della ricorsione.

$$\frac{t[{}^{\textbf{rec}\, x.t}/_x] \to c}{\textbf{rec}\, x.t \to c}$$

Esempio 5.15

$$\textbf{fact} = \textbf{rec}\, f.\lambda x.\,\textbf{if}\; x\; \textbf{then}\; 1\; \textbf{else}\; x \times (f(x-1))$$

$$
\begin{array}{l}
(\textbf{fact}\,2) \to c \\
\Longleftarrow \textbf{fact} \to \lambda x.t \quad t[^2/_x] \to c \\
\Longleftarrow \lambda x.\,\textbf{if}\; x\; \textbf{then}\; 1\; \textbf{else}\; x(\textbf{fact}(x-1)) \to \lambda x.t \quad t[^2/_x] \to c \\
\xLeftarrow{t=\textbf{if}\; x\; \textbf{then}\; 1\; \textbf{else}\; x\times\textbf{fact}(x-1))} \textbf{if}\; 2\; \textbf{then}\; 1\; \textbf{else}\; 2 \times (\textbf{fact}(2-1)) \to c \\
\Longleftarrow 2 \times (\textbf{fact}(2-1)) \to c \\
\xLeftarrow{c=c_1\times c_2} 2 \to c_1 \quad (\textbf{fact}(2-1)) \to c_2 \\
\xLeftarrow{c_1=2} \textbf{fact} \to \lambda x.t \qquad \underbrace{t[^{2-1}/_x] \to c_2}_{2-1 \text{ non viene valutato (lazy)}} \\
\Longleftarrow \textbf{if}(2-1)\, \textbf{then}\; 1\; \textbf{else}(2-1) \times (\textbf{fact}((2-1)-1)) \to c_2 \\
\Longleftarrow 2-1 \to n \quad n \neq 0 \quad (2-1) \times \textbf{fact}((2-1)-1) \to c_2 \\
\xLeftarrow{n=n_1+n_2 \quad 2\to n_1 \quad 1\to n_2 \quad n=2-1=1 \quad c_2=c_3\times c_4} 2-1 \to c_3 \quad \textbf{fact}((2-1)-1) \to c_4 \\
\Longleftarrow \textbf{if}\; 2-1\; \textbf{then}\; 1\; \textbf{else}(2-1) \times (fact((2-1)-1)) \to c_4 \\
\xLeftarrow{c_3=1 \quad \ldots altri\ passaggi} (2-1)-1 \to 0 \qquad 1 \to c_4 \\
\xLeftarrow{c_4=1 \quad c_3=1 \quad c_2=c_3\times c_4=1 \quad c=c_1\times c_2\times 1=2} \square
\end{array}
$$

Esempio 5.16 (Esempio di valutazione eager e lazy)

- ***Valutazione lazy***
  La lazy porta il termine sempre in forma canonica, la eager no. La lazy valuta l'espressione solamente se è strettamente necessario valutarla. Ha lo svantaggio di ripetere più volte la valutazione.
  In questo esempio tutto ha tipo int.

$$
\begin{array}{l}
((\lambda x : int.3)\; \textbf{rec}\, y : int.y) : int \\
\Longleftarrow \lambda x : int.3 \to \lambda x.t \quad t[^{\textbf{rec}\, y.y}/_t] \to c \\
\xLeftarrow{+=3} 3[^{\textbf{rec}\, y.y}/_x] \to c \xLeftarrow{c=3} \square
\end{array}
$$

- ***Valutazione eager***
  Valuta subito, correndo il rischio di non terminare l'esecuzione del programma. Lo stesso programma potrebbe terminare con l'uso della valutazione lazy.

$$
\begin{array}{l}
((\lambda x : int.3)\; \textbf{rec}\, y : int.y) : int \\
\Longleftarrow \lambda x : int.3 \to \lambda x.t \quad \textbf{rec}\, y.y \to c_1 \quad t[^{c_1}/_x] \to c \\
\xLeftarrow{+=3} \textbf{rec}\, y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
\xLeftarrow{y=\textbf{rec}\, y.y} \textbf{rec}\, y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
\xLeftarrow{y=\textbf{rec}\, y.y} \ldots
\end{array}
$$

La valutazione quindi in questo caso non termina.

Nel corso verra sempre utilizzata a valutazione lazy.

Teorema 5.17

i) Se $t \to c$ e $t \to c'$ allora $c = c'$

ii) Se $t \to c$ *e* $t : \tau$ *allora* $c : \tau$

TODO: Dimostrazione per induzione sulle regole negli appunti.

6. Complementi di teoria dei domini e semantica denotazionale di HOFL

Passiamo ad introdurre la semantica denotazionale di HOFL. Prima di farlo però abbiamo bisogno di fare alcune considerazioni.

In qualunque linguaggio Turing-equivalente avremo bisogno di calcolare qualche punto fisso per dare la semantica denotazionale. Nel teorema di Kleene (3.45 a pagina 44) abbiamo dato un metodo per calcolare tale punto fisso: questo metodo si basa sul fatto che il dominio semantico deve essere un $CPO_{\perp}$ e che le funzioni che vi calcoliamo sopra devono essere continue.

Se questo ci veniva gratuito in IMP, dove conoscevamo i domini semantici ed eravamo sicuri della loro natura di $CPO_{\perp}$, non possiamo dire la stessa cosa di HOFL: adesso abbiamo infiniti domini visto che ne abbiamo uno per ogni tipo ed i tipi, essendo costruiti ricorsivamente con la regola

$$\tau ::= int \quad | \quad \tau_1 \times \tau_2 \quad | \quad \tau_1 \rightarrow \tau_2$$

sono infiniti.

Per ovviare a questo problema possiamo fare due cose: innanzitutto dare una costruzione dei domini semantici in modo da poter provare per induzione strutturale che formano un $CPO_{\perp}$, grazie anche ad un'operazione detta di *lifting*. Inoltre cercheremo di costruirci delle regole con le quali sviscerare le funzioni che andremo a calcolare su tali CPO e vedere facilmente che sono continue.

Nella prima parte ci occuperemo dunque di teoria dei domini, quindi andremo a vedere in dettaglio la semantica denotazionale.

6.1. Teoria dei domini

Una funzione di denotazione, come abbiamo visto nei capitoli scorsi, associa ad un valore in un dominio sintattico un altro valore all'interno di un dominio semantico. Nella semantica denotazionale di IMP abbiamo definito dunque tre funzioni, una per ogni dominio sintattico del linguaggio:

- la funzione $\mathscr{A}$ associava elementi del dominio sintattico Aexpr a funzioni che prendevano uno stato e restituivano numeri naturali
- la funzione $\mathscr{B}$ associava elementi del dominio Bexpr a funzioni che prendevano uno stato e restituivano valori boolean
- la funzione $\mathscr{C}$ associava comandi a funzioni che prendevano uno stato e restituivano uno stato modificato

Questo non possiamo farlo nel caso di HOFL. In HOFL abbiamo infiniti domini sintattici, essendoci infiniti tipi. Infatti, essi sono definiti dalla produzione

$$\tau ::= int|\tau \times \tau|\tau \rightarrow \tau$$

che genera infiniti risultati. Avremo bisogno dunque di definire una costruzione ricorsiva per costruirci di volta in volta il tipo di cui abbiamo bisogno.

Per farlo dovremo definire, per ogni produzione che genera un tipo τ, il $CPO_{\perp}$ relativo a tale tipo.

6.1.1. $CPO_\perp$ per il tipo *int*

Abbiamo già definito un $CPO_\perp$ per int: è un CPO tale che

- ogni numero *non è* in relazione con gli altri numeri (importante non confondersi con l'ordinamento classico sui naturali!)
- esiste un elemento $\perp$ minore di tutti i numeri
- esiste un elemento $\top$ maggiore di tutti i numeri

Si tratta ovviamente di un CPO: qualunque catena ha un LUB (visto che tutte le catene hanno lunghezza finita, massimo 3), ed esiste un $\perp$.

6.1.2. $CPO_\perp$ per il tipo $\tau_1 \times \tau_2$

6.1.2.1. Costruzione del $CPO_\perp$

Passiamo al caso in cui un tipo è costruito come prodotto di due tipi. Questi due tipi saranno costruiti come

$$\mathscr{D} = (D, \sqsubseteq_D)$$
$$\mathscr{E} = (E, \sqsubseteq_E)$$

In questo caso, posso definire il nuovo tipo

$$\mathscr{D} \times \mathscr{E} = (D \times E, \sqsubseteq_{D\times E})$$

nel quale

- l'insieme è quello delle coppie formate da un elemento di D ed uno di E
- l'ordinamento è tale che $(d_1, e_1) \sqsubseteq_{D\times E} (d_2, e_2) \Leftrightarrow d_1 \sqsubseteq_D d_2 \wedge e_1 \sqsubseteq_E e_2$
- esiste un elemento $\perp_{D\times E} = (\perp_D, \perp_E)$

L'ambiente risultante è ovviamente un $CPO_\perp$: è banale dimostrare che la relazione $\sqsubseteq_{D\times E}$ è riflessiva, transitiva ed antisimmetrica ed abbiamo definito un $\perp$ che ovviamente è bottom della relazione. Rimane da mostrare che è anche completo.

Teorema 6.1 (Completezza del $CPO_\perp$ per $\tau ::= \tau \times \tau$)
Vale

$$\bigsqcup_{i\in\omega}(d_i, e_i) = \left(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i\right) = lub\,\{(d_i, e_i)\}$$

Dimostrazione. è ovvio che $(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i)$ è un maggiorante per la catena (d_i, e_i): dobbiamo dimostrare che è il minimo.

Supponiamo che non sia minimo: dovrebbe esistere (d_k, e_k) in cui $\bigsqcup_{i\in\omega} d_i \sqsubseteq d_k$ oppure $\bigsqcup_{i\in\omega} e_i \sqsubseteq e_k$, il che è per definizione impossibile. ☠

6.1.2.2. L'operatore π

Sul dominio definito nel paragrafo precedente definiamo l'operatore di proiezione, che ci permetterà di passare da una coppia alle sue singole componenti.

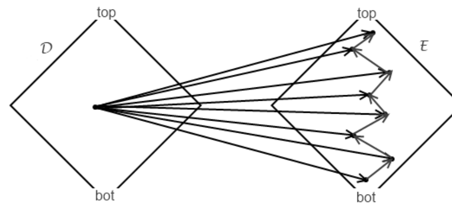


Figura 6.1.: Una catena di funzioni

Definizione 6.2 (Operatore π di proiezione)
Data una coppia $(d, e) \in D \times E$, definisco

- $\pi_1((d, e)) = d$
- $\pi_2((d, e)) = e$

Vediamo la continuità di questo operatore, che ci permetterà di spostare a piacere il simbolo di limite al suo interno o viceversa.

Teorema 6.3 (Continuità di π)
L'operatore di proiezione π è continuo

Dimostrazione.

$$\pi_1\left(\bigsqcup_i (d_i, e_i)\right) = \pi_1\left(\left(\bigsqcup_i d_i\right), \left(\bigsqcup_i e_i\right)\right) = \bigsqcup_i d_i = \bigsqcup_i \pi_1((d_i, e_i))$$

☠

6.1.3. $CPO_\perp$ per il tipo $\tau_1 \to \tau_2$

Vediamo infine il caso in cui il nostro tipo è funzionale: dati due tipi

$$\mathscr{D} = (D, \sqsubseteq_D)$$
$$\mathscr{E} = (E, \sqsubseteq_E)$$

posso definire il nuovo $CPO_\perp$ come

$$[\mathscr{D} \to \mathscr{E}] = ([D \to E], \sqsubseteq_{[D \to E]})$$

dove

- $[D \to E] = \{f | f : D \to E, f$ è continua$\}$ (insieme di tutte le funzioni continue da D ad E)
- $f \sqsubseteq_{[D \to E]} g \Leftrightarrow \forall d \in D. f(d) \sqsubseteq_E g(d)$
- $\perp_{[D \to E]}(d) = \perp_E$

Vediamo un primo teorema che interessa tale dominio, che ci servirà per dimostrarne la continuità.

Lemma 6.4 ($(\bigsqcup_n f_n)(d) = \bigsqcup_n (f_n(d))$)
Vale $(\bigsqcup_n f_n)(d) = \bigsqcup_n (f_n(d))$: il limite di una catena di funzioni applicata ad un argomento d è uguale al limite dela catena dei risultati delle funzioni della catena applicate a d

Dimostrazione. Prendiamo la catena di funzioni nella figura 6.1. Secondo l'ordinamento che abbiamo delineato, ogni funzione nella catena applicata allo stesso elemento di $\mathscr{D}$ genererà un elemento di $\mathscr{E}$ maggiore della funzione precedente. è dunque vero che i risultati della catena di funzioni applicate allo stesso elemento sono a loro volta una catena, indicata con le frecce grige in figura 6.1.

É anche graficamente ovvio che la funzione limite della catena di funzioni applicata a d, qualunque esso sia, è uguale al limite della catena dei risultati in $\mathscr{E}$.

Visto questo lemma, passiamo a dimostrare il teorema

Teorema 6.5 (Completezza dello spazio funzionale)
Vale

$$\left(\bigsqcup_n f_n\right)\left(\bigsqcup_m d_m\right) = \bigsqcup_m \left(\bigsqcup_n f_n(d_m)\right)$$

cioè che la funzione applicata al limite della catena $\bigsqcup_m d_m$ è uguale al limite della catena dei risultati della funzione applicata alla catena d_m.

Dimostrazione. Al solito, si dimostra con una serie di uguaglianze che

$(\bigsqcup_n f_n)(\bigsqcup_m d_m) =$	per il lemma visto sopra
$\bigsqcup_n (f_n(\bigsqcup_m d_m)) =$	perché prendo solamente funzioni continue
$\bigsqcup_n (\bigsqcup_m (f_n(d_m))) =$	come vedremo in seguito
$\bigsqcup_m (\bigsqcup_n (f_n(d_m))) =$	per il lemma visto sopra
$\bigsqcup_m ((\bigsqcup_n f_n)(d_m))$	□

Rimane un dubbio: se una funzione prende come parametro o restituisce una coppia di valori, come ci dobbiamo comportare riguardo alla continuità? Vediamo nelle prossime sezioni due tecniche per ridurre questi casi a normali valutazioni di continuità.

6.1.3.1. Continuità in funzioni del tipo $\tau_1 \to (\tau_2 \times \tau_3)$

Possiamo immaginare una funzione ritorna come risultato una coppia, come il *pairing* di due funzioni

$$(f, g) : S \to D \times E$$

tale che, date le funzioni $f : S \to D$ e $g : S \to E$, vale $(f, g)(s) = (f(s), g(s))$

Teorema 6.6 (Continuità di (f,g))
Date due funzioni continue $f : S \to D$ e $g : S \to E$, la funzione composta $(f, g) : S \to D \times E$ è continua.

Dimostrazione. Si dimostra facilmente che

$$(f, g)\left(\bigsqcup_i s_i\right) = \left(f\left(\bigsqcup_i s_i\right), g\left(\bigsqcup_i s_i\right)\right) = \left(\bigsqcup_i f(s_i), \bigsqcup_i g(s_i)\right) = \bigsqcup_i (f(s_i), g(s_i)) = \bigsqcup_i (f, g)(s_i)$$

come volevasi dimostrare.

Dunque, per dimostrare la continuità di una funzione che restituisce una coppia di valori basta dimostrare la continuità delle funzioni che restituiscono uno dei due valori separato dall'altro.

Definiamo anche la proprietà opposta.

Teorema 6.7 (Continuità delle singole funzioni di una coppia)
Data una funzione composta continua $(f, g) : S \to D \times E$, le funzioni $f : S \to D$ e $g : S \to E$ sono continue.

Dimostrazione. Prendiamo una funzione $h : S \to D \times E$. Questa può essere scomposta in due funzioni

- $f : S \to D = h; \pi_1$
- $g : S \to E = h; \pi_2$

dove $(h; \pi_1)(x) = \pi_1(h(x))$
Sappiamo che sia h che π_1 sono continue. Dimostriamo che l'operatore di concatenazione ; preserva la continuità.

$$(f; g)\left(\bigsqcup_i s_i\right) = g\left(f\left(\bigsqcup_i s_i\right)\right) = g\left(\bigsqcup_i f(s_i)\right) = \bigsqcup_i g(f(s_i)) = \bigsqcup_i (f; g)(s_i)$$

come volevasi dimostrare.

Abbiamo dunque visto che, in presenza di una funzione che restituisce una coppia di elementi, possiamo dividerla ed unirla a piacimento preservando la continuità.

Notare che non esistono funzioni con più di due argomenti: una funzione con tre argomenti viene considerata con una funzione che prende un argomento ed una coppia di argomenti. Le proprietà appena viste dunque sono sufficienti per decomporre tale funzione a funzioni singole, e considerarle separatamente per quel che riguarda la continuità.

Per quanto riguarda la notazione ";" è una simbologia del tutto equivalente alla composizione di funzioni matematiche "∘" ma l'ordine di applicazione va da sinistra a destra invece che da destra a sinistra. Questo risulta comodo in ambito informatico per rappresentare l'applicazione di comandi nell'ordine in cui vengono letti.

$$f \circ g(x) = f(g(x)) \qquad f; g(x) = g(f(x))$$

6.1.3.2. Continuità in funzioni del tipo $D \times E \to S$

Trattiamo adesso il caso in cui abbiamo una funzione che prende una coppia di argomenti.

Per trattare questo caso, dobbiamo fare una premessa: dobbiamo vedere bene il funzionamento di un $CPO_\perp$ sulle coppie. Avendo due ordinamenti separati che si incroceranno, come possiamo vedere nella tabella sottostante

$$\begin{array}{ccccccc}
d_{00} & \sqsubseteq & d_{01} & \sqsubseteq & d_{02} & \sqsubseteq & \cdots \\
\sqcap\!| & & \sqcap\!| & & \sqcap\!| & & \\
d_{10} & \sqsubseteq & d_{11} & \sqsubseteq & d_{12} & \sqsubseteq & \cdots \\
\sqcap\!| & & \sqcap\!| & & \sqcap\!| & & \\
d_{20} & \sqsubseteq & d_{21} & \sqsubseteq & d_{22} & \sqsubseteq & \cdots \\
\sqcap\!| & & \sqcap\!| & & \sqcap\!| & & \\
\vdots & & \vdots & & \vdots & & \ddots
\end{array}$$

In orizzontale ci muoviamo sul primo ordinamento, in verticale sul secondo.

Prendiamo l'insieme $\{d_{nm}\}$, e chiediamoci se tale insieme possiede un LUB. Siamo sul bottom, che sarà d_{00}: la presenza di tale bottom ci è assicurata dal fatto che i due ordinamenti sui quali ci siamo basati hanno entrambi bottom. Muoviamoci sulla prima riga in alto: otterremo una catena che è di fatto equivalente ad una catena sul primo ordinamento. Questa catena avrà dunque un LUB: chiamiamolo $d_{0\bar{k}}$. è importante

capire che questo LUB cadrà nella stessa colonna per tutte le righe: infatti, prendendo la generica catena $\{d_{in_k}\}_k$ abbiamo che questa è equivalente alla catena $\{n_k\}_k$, e che vale $\bigsqcup_k n_k = n_{\bar{k}} \Rightarrow \bigsqcup_k d_{in_k} = n_{i\bar{k}}$. Importante è che questo vale per qualunque i: dunque, se prendiamo la sequenza $\{d_{h\bar{k}}\}_h$ questa sarà ancora una catena equivalente questa volta ad una catena del secondo ordinamento. Troveremo dunque un $\bar{h}$ tale che $\bigsqcup_{nm} \{d_{nm}\} = d_{\bar{h}\bar{k}}$.

Abbiamo dunque visto che possiamo separare le operazioni di limite sulle due componenti della coppia. Ma c'è di più! è dimostrabile che $\bigsqcup_{nm} d_{nm} = \bigsqcup_n = d_n$, cioè che basta prendere il limite degli elementi sulla diagonale. Vediamo la dimostrazione.

Lemma 6.8 (Limite di una catena su una matrice)
Vale

$$\bigsqcup_{nm} d_{nm} = \bigsqcup_{n} = d_n$$

Dimostrazione. Dimostriamo nei due sensi.

è facile vedere che $\bigsqcup_{nm} d_{nm} \sqsupseteq \bigsqcup_k = d_k$, in quanto basta prendere n = m = k.

Vediamo adesso che $\bigsqcup_{nm} d_{nm} \sqsubseteq \bigsqcup_n = d_n$. Supponiamo che d sia il limite di $\bigsqcup_n$, e che dunque $d_{kk} \sqsubseteq d$, e vogliamo che valga $d_{nm} \sqsubseteq d$ per qualunque n ed m. Ma prendendo $k = max\{n, m\}$ abbiamo che $d_{nm} \sqsubseteq d_{kk} \sqsubseteq d$. ☠

Notare che la relazione $\bigsqcup_n \bigsqcup_m d_{nm} = \bigsqcup_m \bigsqcup_n d_{nm}$ è il passo mancante che ci mancava per dimostrare il teorema 6.5.

Siamo adesso pronti ad enunciare finalmente la regola per verificare la continuità nel caso che vogliamo.

Teorema 6.9 (Continuità di una funzione con una coppia come argomento)
Sia $f : D_1 \times D_2 \to E$. Essa è continua se e solo se

$$\forall d_1, d_2. \begin{cases} f_{d_1} : D_2 \to E = \lambda d.f(d_1, d) \text{ è continua} \\ f_{d_2} : D_1 \to E = \lambda d.f(d, d_2) \text{ è continua} \end{cases}$$

Dimostrazione. Dimostriamo solamente la sufficienza.

$$f\left(\bigsqcup_n (x_n, y_n)\right) = f\left(\bigsqcup_n x_n, \bigsqcup_m y_m\right) = \bigsqcup_n f\left(x_n, \bigsqcup_m y_m\right) = \bigsqcup_n \bigsqcup_m f(x_n, y_m) = \bigsqcup_k (x_k, y_k)$$

il primo passaggio è semplicemente distribuzione all'interno della coppia, quindi si utilizzano le ipotesi per portare fuori i limiti considerando parametrici prima y_m poi x_n, quindi si utilizza il lemma 6.8 per unificare l'indice. ☠

6.1.4. Dimostrazioni di continuità

Vediamo alcuni esempi di funzioni interessanti in HOFL, e dimostriamone la continuità.

Esempio 6.10 (Continuità della funzione apply)
La funzione apply di HOFL è una funzione

$$apply : [D \to E] \times D \to E$$

che ha la seguente semantica: $apply(f, d) = f(d)$

Vediamo che è continua (e che dunque apply : $[[D \to E] \times D \to E]$*), utilizzando il teorema 6.9: dobbiamo dimostrare che è continua sui due argomenti. Prendendo d come parametro abbiamo, assumendo che le* f_n *siano continue,*

$$apply\left(\bigsqcup_n f_n, d\right) = \left(\bigsqcup_n f_n\right)(d) = \bigsqcup_n (f_n(d)) = \bigsqcup_n apply(f_n, d) \quad \square$$

Prendendo invece f come parametro, ed assumendo la continuità di d_m *ho*

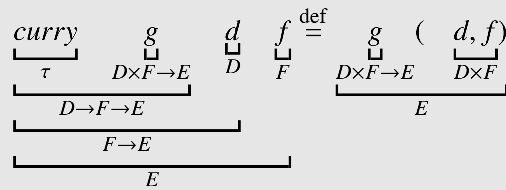
$$apply\left(f, \bigsqcup_m d_m\right) = \left(f\left(\bigsqcup_m d_m\right)\right) = \bigsqcup_m (f(d_m)) = \bigsqcup_m apply(f, d_m) \quad \square$$

Per il teorema 6.9 posso dunque dire che la funzione apply è continua.

Esempio 6.11 (Funzioni Curry ed Uncurry)
La funzione curry è una funzione con tipo

$$curry : \tau \quad \tau = (D \times F \to E) \to (D \to (F \to E))$$

tale che curry g d f $\stackrel{\text{def}}{=} g(d, f)$*. Tipiamo questo termine per comprendere meglio le grandezze in gioco.*



TODO: aggiungere definizione fix e prova continuità

6.1.5. Lifting

Abbiamo visto finora come costruire una gerarchia di categorie sintattiche a partire dalla categoria di base degli interi. Ogni combinazione di interi, funzioni e coppie ha un proprio CPO, costruito ricorsivamente a partire dal suo tipo di base.

Tuttavia, questo non ci basterà. Vedremo che a volte per dimostrare l'equivalenza fra la semantica denotazionale e quella operazionale ci servirà un'ulteriore passo: l'aggiunta di un $\perp$ particolare per ogni categoria sintattica. è un operazione che viene detta di *lifting*.

Vediamo di capire cosa questo comporti. Mettiamo di aver costruito un tipo come composizione di altri due tipi. Ci ritroviamo con il nostro CPO $\mathscr{D} = (D, \sqsubseteq_D)$. Costruiamo su di esso un nuovo CPO $\mathscr{D}_\perp = (D_\perp, \sqsubseteq_{D_\perp})$ tale che

- $D_\perp = \{1\} \times D \quad \cup \quad \{0\} \times \{\perp\}$
- $d_1 \sqsubseteq_D d_2 \quad \to \quad (1, d_1) \sqsubseteq_{D_\perp} (1, d_2)$
- $\forall d \in D_\perp .\ \perp_{D_\perp} \sqsubseteq_{D_\perp} d \quad \text{dove } \perp_{D_\perp} = (0, \perp)$

Può sembrare una inutile complicazione ma vedremo che sarà necessario per provare l'equivalenza tra semantica operazionale e denotazionale 7.2. D'altra parte, avevamo già fatto una cosa simile con la trasformazione dell'insieme Σ nell'insieme $\Sigma_\perp$ per gestire i comandi che non terminavano.

Definizione 6.12 (Operatore di lifting)
è un operatore $\lfloor\rfloor : D \to D_\perp$ *tale che*

$$\lfloor d \rfloor = (1, d)$$

Al solito, dunque, definiremo per ogni funzione $f : D \to E$ una nuova funzione f^* tale che $f^* : D_\perp \to E$

$$f^*(d_\perp) = \textbf{case} \quad d_\perp \quad \perp_{D_\perp} \to \perp_E \quad \lfloor d \rfloor \to f(d)$$

La stellina controlla l'argomento della funzione, se è bottom ritorna bottom altrimenti lo de-lifta e lo passa alla funzione reale.

Infine definiamo l'operatore che useremo direttamente nella nostra semantica denotazionale, che fa da abbreviazione per l'operatore stellina nel caso in cui vogliamo sostituire l'espressione da valutare in un'altra.

Definizione 6.13 (Operatore `let`)

$$let\ x \Leftarrow d_\perp.\ e = \underbrace{\underbrace{\underbrace{(\lambda x.\ e)}_{D\to E}{}^*}_{D_\perp\to E}\ (d_\perp)}_{E} = \begin{array}{ll} case & d_\perp \\ & \perp_{D_\perp} \to \perp_E \\ & \lfloor d \rfloor \to e[^d/_x] \end{array}$$

Possiamo vederlo in due modi equivalenti:

- il let estende il tipo della funzione da applicare tramite il lifting ($*$)
- il let valuta l'argomento e se differente da $\perp_{D_\perp}$, opera un *de-lifting* e passa l'argomento alla funzione.

6.2. Semantica denotazionale di HOFL

Dopo questa lunga premessa, siamo finalmente in grado di dare la semantica denotazionale di HOFL. Avremo un'unica funzione di valutazione semantica, avendo un'unica categoria sintattica (quella dei termini):

$$[\![t : \tau]\!] : Env \to (V_\tau)_\perp$$

Ovviamente, anche se i termini sono di una sola categoria sintattica potremo avere infinite categorie semantiche in uscita: per questo, il codominio della funzione sarà parametrizzato in base al tipo del termine che stiamo valutando.

Definiamo dunque, per un qualunque dominio τ, un dominio semantico $(V_\tau)_\perp = V_\tau \cup \perp_\tau$ che contiene tutti i valori che può assumere il tipo τ più il solito valore bottom.

Avremo anche qua bisogno di un ambiente Env: il nostro ambiente sarà $\rho : Var \to \bigcup_\tau (V_\tau)_\perp$, cioè una funzione da nomi di variabili ad un elemento di *qualunque dominio semantico esistente*.

Come abbiamo visto nella sezione sulla teoria dei domini, possiamo definire tali domini semantici nella seguente maniera

$$\begin{aligned} (V_{int})_\perp &= \mathbb{N}_\perp \\ (V_{\tau_1\times\tau_2})_\perp &= ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ (V_{\tau_1\to\tau_2})_\perp &= [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp \end{aligned}$$

Questi domini semantici continuano ad essere dei CPO, condizione fondamentale per poi calcolarci il punto fisso.

Vediamo adesso in dettaglio la semantica denotazionale per ogni categoria sintattica. Inoltre controlleremo una cosa importante: tutte le funzioni di base che utilizzeremo dovranno essere rigorosamente continue. In caso contrario, non potremmo calcolarne il punto fisso che ci servirà per gestire correttamente la ricorsione.

6.2.1. Costanti

Definiamo la semantica denotazionale delle variabili come

$$[\![n]\!]\rho = \lfloor n \rfloor$$

Tipi Tipiamo le due espressioni eguagliate. La prima è

$$\underbrace{[\![\underbrace{n}_{int}]\!]\rho}_{(V_{int})_\perp}$$

Valutare un'espressione $n : int$ porta ad avere un risultato nell'insieme $(V_{int})_\perp$. Passando alla seconda metà dell'espressione

$$\underbrace{\lfloor \underbrace{n}_{int} \rfloor}_{(V_{int})_\perp}$$

semplicemente prendiamo n, di tipo int, e lo liftiamo facendolo diventare di tipo $\mathbb{N}_\perp$, che come abbiamo visto precedentemente è uguale a $(V_{int})_\perp$.

6.2.2. Variabili

Definiamo la semantica denotazionale delle costanti come

$$[\![x]\!]\rho = \rho x$$

Tipi Tipiamo le due espressioni eguagliate. La prima è

$$\underbrace{[\![\underbrace{x}_{\tau}]\!]\rho}_{(V_\tau)_\perp}$$

Questa volta valutiamo un argomento di un tipo qualunque: chiamiamolo τ. Valutandolo otteniamo mediante la funzione di valutazione semantica applicata all'ambiente ρ un elemento in $(V_\tau)_\perp$. Dall'altra parte invece

$$\underbrace{\rho \underbrace{x}_{\tau}}_{(V_\tau)_\perp}$$

applichiamo la funzione ρ ad x ed otteniamo un elemento in $(V_\tau)_\perp$ come dall'altro lato.

6.2.3. Operazioni binarie

Passiamo a definire la semantica denotazionale delle operazioni binarie. Le cose si fanno leggermente più complesse.

$$[\![t_1 \textbf{ op } t_2]\!]\rho = [\![t_1]\!]\rho \; \underline{op}_\perp \; [\![t_2]\!]\rho$$

dove

$$\underline{op}_\perp(x_1, x_2) = \begin{cases} \lfloor x_1 \underline{op} x_2 \rfloor & \text{se } \lfloor x_1 \rfloor \neq \perp_{\mathbb{N}_\perp} \wedge \lfloor x_2 \rfloor \neq \perp_{\mathbb{N}_\perp} \\ \perp_{\mathbb{N}_\perp} & \text{altrimenti} \end{cases}$$

Tipi Il tipaggio ovviamente si fa più complesso, ma rimane comunque comprensibile. Nella parte sinistra abbiamo

$$\underbrace{[\![t_1 \ \mathbf{op} \ t_2]\!]\rho}_{(V_{int})_\perp = N_\perp}$$

Semplicemente è la valutazione di un'operazione binaria op che restituisce un intero: il tipo restituito dalla valutazione semantica è dunque $(V_{int})_\perp$. Nella parte destra invece abbiamo

$$\underbrace{\underbrace{[\![t_1]\!]\rho}_{(V_{int})_\perp = N_\perp} \quad \underbrace{\underline{op}_\perp}_{N_\perp \times N_\perp \to N_\perp} \quad \underbrace{[\![t_2]\!]\rho}_{(V_{int})_\perp = N_\perp}}_{N_\perp}$$

Valutando t_1 e t_2 otteniamo due oggetti di tipo $(V_{int})_\perp$. La funzione $\underline{op}_\perp$ è una funzione che va da coppie di elementi di $(V_{int})_\perp$ ad elementi dello stesso insieme, quindi anche tutta la parte destra ha tipo $(V_{int})_\perp$.

Continuità Trattandosi di una funzione di due termini, dobbiamo dimostrare per ricorsione strutturale che se i due termini ai quali la applichiamo sono continui anche la funzione rimane continua.

Teorema 6.14 (Continuità di $\underline{op}$)
L'operatore $\underline{op}$ è continuo se sono continui i suoi argomenti.

Dimostrazione. Lavoriamo su due elementi di partenza nel CPO degli interi, che come abbiamo visto numerose volte è un CPO piatto. Questo significa che tutte le catene che posso costruire sulle sue coppie hanno lunghezza massima 3 prima di appiattirsi sul limite. Ci basta dunque dimostrarne la monotonia.

$$(x'_1, x'_2) \sqsubseteq (x''_1, x''_2) \quad \Rightarrow \quad x'_1 \underline{op} x'_2 \sqsubseteq x''_1 \underline{op} x''_2$$

Essendo una funzione a due argomenti possiamo applicare il teorema 6.9 che ci permette di poter considerare i due parametri in maniera separata.

Poniamo dunque $x'_2 = x''_2 = \bar{x}$ e vediamo la monotonia solamente in base a x'_1 ed x''_1.

C'è innanzitutto da notare che se prendiamo $\bar{x} = \perp_{\mathbb{N}_\perp}$ la conseguenza dell'implicazione è immediatamente vera in quanto, per qualunque scelta possiamo fare di x'_1 ed x''_1, avremo sempre e comunque $\perp \sqsubseteq \perp$ che è una tautologia.

Prendendo invece $\bar{x} = n$, avremo che $(x'_1, n) \sqsubseteq (x''_1, n) \Leftrightarrow x'_1 \sqsubseteq x''_1$ nei rispettivi ordinamenti. Quindi può succedere che

- $x'_1 = x''_1 = \perp$, il che porterebbe all'eguaglianza $\perp \sqsubseteq \perp$
- $x'_1 = \perp$ e $x''_1 = m$, il che porterebbe a $\perp \sqsubseteq \lfloor n + m \rfloor$ che è sicuramente vera per qualunque scelta di n ed m
- $x'_1 = x''_1 = m$, il che porterebbe all'eguaglianza $\lfloor n + m \rfloor \sqsubseteq \lfloor n + m \rfloor$

A costo di essere ripetitivi, ribadiamo che l'ordinamento degli interi non contempla confronti fra interi diversi.

6.2.4. Condizionale

La semantica denotazionale del comando if è

$$[\![\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2]\!]\rho = Cond\,([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

dove

$$Cond_\tau(z_0, z_1, z_2) = \begin{cases} z_1 & \text{se } z_0 = \lfloor 0 \rfloor \\ z_2 & \text{se } z_0 = \lfloor n \rfloor \quad n \neq 0 \\ \perp_{(V_\tau)_\perp} & \text{se } z_0 = \perp_{\mathbb{N}_\perp} \end{cases}$$

Tipi Vediamo il tipaggio della parte sinistra. Abbiamo che

$$\underbrace{\llbracket \underbrace{\textbf{if } \underbrace{t_0}_{int} \textbf{ then } \underbrace{t_1}_{\tau} \textbf{ else } \underbrace{t_2}_{\tau}}_{\tau} \rrbracket \rho}_{(V_\tau)_\bot}$$

Il comando if ha un intero come guardia e due elementi dello stesso tipo τ come rami then ed else. Il comando restituisce un elemento di tipo τ, quindi la sua semantica denotazionale restituisce un elemento del tipo $(V_\tau)_\bot$.

Vediamo invece la parte destra.

$$\underbrace{Cond(\underbrace{\llbracket t_0 \rrbracket}_{(V_{int})_\bot} \underbrace{\llbracket t_1 \rrbracket}_{(V_\tau)_\bot} \underbrace{\llbracket t_2 \rrbracket}_{(V_\tau)_\bot})}_{(V_\tau)_\bot}$$

è molto semplice: si tratta di applicare il comando Cond, il quale restituisce elementi del tipo del suo secondo e terzo argomento.

Continuità Anche qua dobbiamo dimostrare che la funzione Cond è continua.

Teorema 6.15 (Continuità di Cond)
La funzione Cond è continua

Dimostrazione. Per il teorema 6.9 possiamo ancora analizzare la continuità della funzione analizzandone un argomento alla volta.

Prendendo due parametri a e b, analizziamo il comportamento di

$$Cond_{b,c}(z_0) = Cond(z_0, a, b)$$

è una funzione che varia su un parametro intero, quindi il dominio è piatto e possiamo permetterci di dimostrare solamente la monotonia: questa implicherà automaticamente la continuità. è facile vedere che la continuità viene preservata: $Cond_{a,b}(\bot)$ restituisce sempre $\bot$, che sarà sempre minore o uguale rispetto a qualunque altro risultato.

Prendendo invece

$$Cond_{a,c}(z_1) = Cond(a, z_1, c)$$

studiamo il comportamento della funzione al variare di un elemento definito su un CPO con catene potenzialmente infinite: dovremo dimostrare la continuità dunque. Per nostra fortuna è facile farlo. Facciamo variare il primo parametro:

- se $a = \bot$, la nostra funzione restituirà sempre $\bot$. Questo significa che la funzione si riduce ad una funzione costante che, ovviamente, sarà continua.
- se $a = \lfloor 0 \rfloor$, otteniamo la funzione identità, anche lei ovviamente continua
- se $a = \lfloor n \rfloor$, otteniamo un'altra funzione costante che restituisce c per ogni z_1.

La funzione è dunque continua per qualunque valore di z_1. Il ragionamento facendo variare solo z_2 è assolutamente analogo. ☠

6.2.5. Coppie

Diamo la semantica delle coppie.

$$\llbracket (t_1, t_2) \rrbracket \rho = \lfloor (\llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \rfloor$$

Tipi Il tipaggio è semplice. A sinistra abbiamo

$$\underbrace{[\![\underbrace{(\underbrace{t_1}_{\tau_1}, \underbrace{t_2}_{\tau_2})}_{\tau_1\times\tau_2}]\!]\rho}_{(V_{\tau_1\times\tau_2})_\perp}$$

Si commenta da sola. La parte destra è leggermente più complessa ma anche essa autoesplicativa.

$$\underbrace{\lfloor \underbrace{(\underbrace{[\![t_1]\!]\rho}_{(V_{\tau_1})_\perp}, \underbrace{[\![t_2]\!]\rho}_{(V_{\tau_2})_\perp})}_{(V_{\tau_1})_\perp\times(V_{\tau_2})_\perp}\rfloor}_{((V_{\tau_1})_\perp\times(V_{\tau_2})_\perp)_\perp}$$

L'uguaglianza fra tipi vale, contando che per definizione $(V_{\tau_1\times\tau_2})_\perp = ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp$

Continuità La continuità ci viene assicurata dalla definizione dell'operazione di coppia nella sezione 6.1.2.1.

6.2.6. Selezionatori di coppia

La semantica dei selezionatori di coppia è

$$[\![fst(t)]\!]\,\rho = \textbf{let}\, v \Leftarrow [\![t]\!]\,\rho. \quad \pi_1 v$$
$$[\![snd(t)]\!]\,\rho = \textbf{let}\, v \Leftarrow [\![t]\!]\,\rho. \quad \pi_2 v$$

Notare che grazie al `let` l'espressione t viene valutata, de-liftata e sostituita nell'espressione πv, naturalmente se la valutazione di t fallisce, viene restituito il relativo bottom.

Tipi Essendo le due espressioni analoghe, tipiamo solo la prima delle due. A sinistra abbiamo semplicemente

$$\underbrace{[\![\underbrace{fst(\underbrace{t}_{\tau_1\times\tau_2})}_{\tau_1}]\!]\rho}_{(V_{\tau_1})_\perp}$$

mentre a destra abbiamo un'espressione vagamente più complessa.

$$\textbf{let}\ \underbrace{v}_{(V_{\tau_1})_\perp\times(V_{\tau_2})_\perp} \Leftarrow \underbrace{[\![t]\!]\rho}_{(V_{\tau_1\times\tau_2})_\perp}.\ \underbrace{\underbrace{\pi_1}_{(V_{\tau_1})_\perp\times(V_{\tau_2})_\perp\to(V_{\tau_1})_\perp}\ v}_{(V_{\tau_1})_\perp}$$

A v viene associata una coppia di tipo $(V_{\tau_1})_\perp \times (V_{\tau_2})_\perp$ che è il de-liftato di $((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp$ che per definizione è uguale a $(V_{\tau_1\times\tau_2})_\perp$, proprio il tipo di ottenuto dalla valutazione di t. Alla coppia viene poi applicato l'operatore π_1 che la proietta sul primo argomento.

Continuità Anche qua la continuità ci viene gratuita, in quanto gli operatori π la preservano, come visto nel paragrafo 6.1.2.2. Per il resto, si tratta di applicare tale operatore ad una forma canonica.

6.2.7. Lambda function

Il discorso si fa decisamente più complesso per le lambda functions, non tanto per la semantica o il tipaggio, quanto per la dimostrazione della continuità. Vediamo intanto che

$$[\![\lambda x.t]\!]\,\rho = \left\lfloor \lambda d.\, [\![t]\!]\,\rho[^d/_x] \right\rfloor$$

Tipi Vediamo la parte sinistra, che è semplicemente

$$\underbrace{\underbrace{[\![\lambda \underbrace{x}_{\tau_1} . \underbrace{t}_{\tau_2}]\!]}_{\tau_1 \to \tau_2} \rho}_{(V_{\tau_1 \to \tau_2})_\perp}$$

Si costruisce il tipo $\tau_1 \to \tau_2$ della funzione, quindi se ne valuta la semantica che è di tipo $(V_{\tau_1 \to \tau_2})_\perp$. Facilissimo!

Più rognosa è la valutazione del tipo della seconda parte.

$$\underbrace{\lfloor \underbrace{\lambda \underbrace{d}_{(V_{\tau_1})_\perp} . \underbrace{[\![t]\!]\rho[^d/_x]}_{(V_{\tau_2})_\perp}}_{[(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]} \rfloor}_{[(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp}$$

Si tratta del lifting di una funzione che va dal dominio semantico di un valore d, il quale verrà passato per parametro in caso di applicazione di funzione, alla valutazione di un termine in una memoria modificata da tale d. Questa funzione restituisce ciò che restituirebbe t.
Il risultato è un liftato perché non ritorna mai $\perp$

Continuità Dimostrare la continuità di questa funzione non è banale, in quanto non dipende da sottotermini del termine di cui vogliamo dimostrare la continuità ma da un termine valutato in una memoria che non è quella di partenza. Serve una dimostrazione decisamente più robusta[1].

Teorema 6.16 (Continuità della semantica della lambda expression)
La funzione $\lambda d. [\![t]\!] \rho[^d/_x]$ di tipo $(V_\tau)_\perp \to (V_\sigma)_\perp$ dove $d : \tau$ e $t : \sigma$ è continua.

Dimostrazione. La dimostrazione è data per induzione strutturale su t: si può vedere che, in qualunque maniera costruiamo questo termine, questa funzione è sempre continua.

Costante La funzione $\lambda d. [\![n]\!] \rho[^d/_x] = \lambda d.n$ è semplicemente la funzione costante: è ovviamente continua.

Variabile La funzione $\lambda d. [\![x]\!] \rho[^d/_x] = \lambda d.d$ è semplicemente la funzione identità: è ovviamente continua.

Operazione binaria La funzione $\lambda d. [\![t_1 op t_2]\!] \rho[^d/_x] = \lambda d. [\![t_1]\!] \rho[^d/_x] \underline{op}_\perp [\![t_2]\!] \rho[^d/_x]$ è una composizione di funzioni continue, ed è quindi ovviamente continua.

Condizionale Continuo per lo stesso motivo dell'operazione binaria.

Operazioni sulle coppie Continue per lo stesso motivo dell'operazione binaria.

Lambda function La funzione $\lambda d. [\![\lambda x.t]\!] \rho[^d/_y]$ è continua?

- Se x=y è ovviamente continua
- Altrimenti dobbiamo procedere per induzione. Sappiamo per ipotesi induttiva che $\lambda d, d'. [\![t]\!] \rho[^{d'}/_x, ^d/_y]$ è continua. Ma allora per composizione di funzioni lo è anche $curry(\lambda d, d'. [\![t]\!] \rho[^{d'}/_x, ^d/_y]) = \lambda d. \lambda d'. [\![t]\!] \rho[^d/_y][^{d'}/_x] = \lambda d. [\![\lambda x.t]\!] \rho[^d/_y]$, come volevasi dimostrare.

Ricorsione Il caso della ricorsione è analogo a quello della lambda function, applicando anche le funzioni fix ed apply.

Abbiamo dunque visto che, qualunque sia la natura di t, la funzione vista sopra è continua.

[1] NB: questa dimostrazione *non è* stata spiegata a lezione, quindi non è necessario studiarla per l'esame

6.2.8. Applicazione di funzione

Vediamone la semantica.

$$[\![(t_1\ t_2)]\!] = \mathbf{let}\, \varphi \Leftarrow [\![t_1]\!]\, \rho. \varphi([\![t_2]\!]\, \rho)$$

Tipi Il tipaggio è abbastanza semplice.

$$\underbrace{[\![(\underbrace{t_1}_{\tau_1\to\tau_2}\ \underbrace{t_2}_{\tau_1})]\!]}_{(V_{\tau_2})_\perp}$$

La prima metà è un'applicazione di una funzione da τ_1 a τ_2 ad un argomento di tipo τ_1, il cui risultato viene liftato. Nella seconda metà a abbiamo

$$\mathbf{let}\ \underbrace{\varphi}_{[(V_{\tau_1})_\perp\to(V_{\tau_2})_\perp]} \Leftarrow \underbrace{[\![t_1]\!]\rho}_{(V_{\tau_1\to\tau_2})_\perp} . \underbrace{\varphi(\underbrace{[\![t_2]\!]\rho}_{(V_{\tau_1})_\perp})}_{(V_{\tau_2})_\perp}$$

La funzione φ è ottenuta valutando t_1, quindi è del tipo del dominio semantico corrispondente al suo tipo $((V_{\tau_1\to\tau_2})_\perp)$. Essa viene applicata ad un valore di tipo $(V_{\tau_1})_\perp$ per ottenere un elemento di tipo $(V_{\tau_2})_\perp$, che era quello che volevamo.

Continuità L'esecuzione della semantica denotazionale dell'applicazione di funzione è analoga all'esecuzione di una funzione *apply*, vista nell'esempio 6.10. Visto che apply è continua, lo è anche la nostra semantica.

6.2.9. Ricorsione

La semantica della ricorsione è

$$[\![rec\ x.t]\!]\, \rho = \mathbf{fix}\, \lambda d.\ [\![t]\!]\, \rho[^d/_x]$$

Tipi Tipiamo la parte sinistra. Abbiamo

$$\underbrace{[\![rec\ \underbrace{x}_{\tau} . \underbrace{t}_{\tau}]\!]\rho}_{(V_\tau)_\perp}$$

Tutte le componenti del comando rec hanno tipo τ, dunque il risultante è di tipo $(V_\tau)_\perp$. Nella parte destra invece

$$\underbrace{\underbrace{fix}_{[(V_\tau)_\perp\to(V_\tau)_\perp]\to(V_\tau)_\perp}\ \underbrace{\underbrace{\lambda d}_{(V_\tau)_\perp} . \underbrace{[\![t]\!]\rho[^d/_x]}_{(V_\tau)_\perp}}_{[(V_\tau)_\perp\to(V_\tau)_\perp]}}_{(V_\tau)_\perp}$$

Notare che punto fisso trovato è il *minimo punto fisso* (forse sarebbe più corretto usare lfp invece di fix), al contrario di quanto detto per la semantica operazionale, qui otteniamo sempre il minimo.

6.3. La semantica denotazionale è composizionale

Possiamo dimostrare il seguente lemma

Teorema 6.17 (Lemma di sostituzione)
Vale

$$\left[\!\left[t[^{t'}/_x]\right]\!\right]\rho = [\![t]\!]\,\rho[^{[\![t']\!]\rho}/_x]$$

L'importanza di questo teorema è che ci permette di dire che la nostra semantica denotazionale è *composizionale*: se c'è una variabile all'interno di un certo contesto, di ciò che metto all'interno del buco posso guardare anche solamente la semantica denotazionale per avere tutte le informazioni che mi interessano.

$$[\![t_1]\!]\rho = [\![t_2]\!]\rho \Rightarrow \left[\!\!\left[t[{}^{t_1}/{}_{x}]\right]\!\!\right]\rho = \left[\!\!\left[t[{}^{t_2}/{}_{x}]\right]\!\!\right]\rho$$

In questo caso sostituisco t_1 con t_2 senza sapere come sotto fatti dentro, tanto sono garantito che hanno la stessa semantica denotazionale e questo è sufficiente.

7. Equivalenza fra le semantiche di HOFL

La dimostrazione dell'equivalenza fra la semantica operazionale e la semantica denotazionale di HOFL è una dimostrazione molto complessa. Se nel caso di IMP siamo riusciti a dimostrare la *completa equivalenza* fra le due semantiche (cioé che se una delle due terminava, anche l'altra terminava esattamente con lo stesso risultato) qua non potremo fare la stessa cosa e ci dovremo accontentare di un risultato parziale.

Vedremo innanzitutto le basi dalle quali partiremo per la prova, quindi proveremo ad eseguire la stessa prova che abbiamo effettuato nel caso di IMP e capiremo perché le conclusioni alle quali arriviamo sono meno strette.

7.1. Basi del confronto

Vogliamo confrontare due cose decisamente diverse. Da una parte abbiamo la semantica operazionale, che consiste in un insieme di formule ben formate della forma $t \to c$, in cui t è un termine del linguaggio e c é una cosiddetta forma canonica, assieme ad un sistema di transizione utilizzato per prendere il sottoinsieme di tali regole che sono teoremi di tale sistema. Questo sistema opera su termini *chiusi*, in quanto é una valutazione di esecuzione e non possono mancare dati necessari affinché questa esecuzione vada a buon fine.

Dall'altra parte abbiamo la semantica denotazionale, che ci fornisce una funzione da un ambiente ad un elemento di un insieme $(V_\tau)_\perp$. La semantica denotazionale opera anche su termini *aperti*, in quanto alcuni dati possono essere introdotti attraverso la memoria ρ sulla quale la semantica viene valutata.

Cominciamo ad introdurre un po' di notazione. Indicheremo il fatto che la valutazione operazionale di un termine t termina con il simbolo $t \downarrow$ (il che sarà vero se e solo se $\exists c. t \to c$), mentre la stessa cosa per la semantica denotazionale verrà indicata con $t \Downarrow$ (il che sarà vero se e solo se $[\![t]\!]\rho \neq \perp_{(V_\tau)_\perp}$). Il confronto verrà effettuato solamente sui termini sui quali entrambe le semantiche lavorano, cioè i termini chiusi. La memoria ρ sarà dunque presente ma non utilizzata all'interno delle dimostrazioni.

Riusciremo a dimostrare che $t \downarrow \Leftrightarrow t \Downarrow$, ma non riusciremo a dimostrare che

$$t \to c \quad t' \to c' \quad [\![t]\!] = [\![t']\!] \quad \Rightarrow \quad c = c'$$

In pratica, è possibile che due termini con stessa semantica denotazionale restituiscano due forme canoniche differenti, a meno che non siano di tipo int. Torneremo su questo aspetto nella sezione 7.5.

7.2. Uso del lifting

Come abbiamo visto nel capito sulla teoria dei domini di HOFL 6.1.5, ogni dominio sul quale lavoriamo è un $CPO_\perp$, che si tratti del dominio piatto dei naturali o del frutto di varie operazioni di composizione funzionale e prodotto cartesiano.

In seguito, nonostante ci fosse già sempre un elemento bottom, abbiamo introdotto con il lifting un nuovo bottom ottenedo quindi sia $\perp_{V_\tau}$ che $\perp_{(V_\tau)_\perp}$, a che scopo?

Il problema è che la semantica operazionale non sempre termina restituendo una forma canonica mentre la denotazionale, essendo definita per induzione strutturale, termina sempre e questo invaliderebbe il risultato, che dimostreremo fra poco, che la terminazione operazionale implica la terminazione denotazionale e viceversa. Tuttavia il problema si risolve semplicemente dato che quando non c'è forma canonica il risultato della denotazionale è un $\perp$, esattamete come in IMP.

Putroppo però esistono esistono dei termini, come ad esempio

$$\lambda x.\ rec\ y.\ y$$

che hanno forma canonica ma in denotazionale ritornano $\perp_{V_\tau}$.
In questi casi per distinguere i due tipi di $\perp$, quelli con forma canonica e quelli senza, entra in gioco il lifting: ogni volta che la semantica denotazionale ritorna un $\perp_{V_\tau}$, questo viene liftato in $\lfloor \perp_{V_\tau} \rfloor$ in modo da diventare un normale valore $d_\perp$ del dominio, diverso da $\perp_{(V_\tau)_\perp}$, perfettamente in accordo con la definizione di $\Downarrow$.

Esempio 7.1 (Distinzione dei $\perp$)
Vediamo la semantica denotazionale dei due sequenti termini di cui il primo è una forma canonica mentre il secondo no.

$$\begin{aligned} [\![\lambda x : int.\ rec y : int.\ y]\!] &= \lfloor \perp_{[N_\perp \to N_\perp]_\perp} \rfloor = \perp_{[N_\perp \to N_\perp]} \\ [\![rec x : int \to int.\ x]\!] &= \perp_{[N_\perp \to N_\perp]_\perp} \end{aligned}$$

In questo caso il dominio delle funzioni da int a int possiede il suo bottom $\lambda x. \perp$ e il bottom aggiunto con il lifting.
Se non utilizzassimo il lifting i due termini risulterebbero identici per la semantica denotazionale ma differenti per l'operazionale.

7.3. La terminazione operazionale implica la terminazione denotazionale

Come primo passo, dimostriamo che $t \to c \Rightarrow [\![t]\!]\rho = [\![c]\!]\rho$. Al solito lo dimostreremo per induzione sulle regole.

La proposizione che vogliamo dimostrare è

$$P(t \to c) \stackrel{\text{def}}{=} [\![t]\!]\rho = [\![c]\!]\rho$$

7.3.1. Forma canonica

La prima regola di hofl è l'unico assioma del linguaggio

$$\frac{}{c \to c}$$

É immediato vedere che $P(c \to c) \stackrel{\text{def}}{=} [\![c]\!]\rho = [\![c]\!]\rho$ è ovviamente verificata.

7.3.2. Operazioni binarie

Prendiamo la regola

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \underline{op} n_2}$$

Dobbiamo dimostrare

$$P(t_1 \,\mathbf{op}\, t_2 \to n_1 \underline{\mathbf{op}} n_2) \stackrel{\text{def}}{=} [\![t_1 \,\mathbf{op}\, t_2]\!]\rho = \left[\!\!\left[n_1 \underline{\mathbf{op}} n_2\right]\!\!\right]\rho$$

sotto l'ipotesi di aver già provato che $P(t_1 \to n_1) \stackrel{\text{def}}{=} [\![t_1]\!] = [\![n_1]\!]$ e $P(t_2 \to n_2) \stackrel{\text{def}}{=} [\![t_2]\!] = [\![n_2]\!]$.

Facilmente vediamo che

$[\![t_1 \mathbf{op}\, t_2]\!]\rho =$	per definizione
$[\![t_1]\!]\rho \underline{\mathbf{op}}_\perp [\![t_2]\!]\rho =$	per ipotesi induttiva
$[\![n_1]\!]\rho \underline{\mathbf{op}}_\perp [\![n_2]\!]\rho =$	per definizione
$\lfloor n_1 \rfloor \underline{\mathbf{op}}_\perp \lfloor n_2 \rfloor =$	per definizione dell'operatore
$\lfloor n_1 \underline{op} n_2 \rfloor =$	per definizione
$[\![n_1 \underline{op} n_2]\!]$	□

7.3.3. Condizionale

Prendiamo la regola del condizionale nel caso in cui si segue il ramo *then*.

$$\frac{t_0 \to 0 \quad t_1 \to c_1}{\mathbf{if}\, t_0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2 \to c_1}$$

Dobbiamo dimostrare

$$P(\mathbf{if}\, t_0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2 \to c_1) \stackrel{\text{def}}{=} [\![\mathbf{if}\, t_0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2 \to c_1]\!]\rho = [\![c_1]\!]\rho$$

sotto l'ipotesi di aver già provato che $P(t_0 \to 0) \stackrel{\text{def}}{=} [\![t_0]\!]\rho = [\![0]\!]\rho$ e $P(t_1 \to c_1) \stackrel{\text{def}}{=} [\![t_1]\!]\rho = [\![c_1]\!]\rho$.

Altrettanto facilmente vediamo che

$[\![\mathbf{if}\, t_0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2]\!]\rho =$	per definizione
$Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho) =$	per ipotesi induttiva
$Cond([\![0]\!]\rho, [\![c_1]\!]\rho, [\![t_2]\!]\rho) =$	per definizione
$Cond(\lfloor 0 \rfloor, [\![c_1]\!]\rho, [\![t_2]\!]\rho) =$	per definizione di Cond
$[\![c_1]\!]\rho$	□

Il caso in cui viene eseguito il caso else è assolutamente analogo.

7.3.4. Selettore da coppia

Prendiamo la regola del first

$$\frac{t \to (t_1, t_2) \quad t_1 \to c_1}{\mathbf{fst}(t) \to c_1}$$

Dobbiamo dimostrare $P(\mathbf{fst}(t) \to c_1) \stackrel{\text{def}}{=} [\![fst(t)]\!]\rho = [\![c_1]\!]\rho$ sotto l'ipotesi di aver già provato $P(t_1 \to c_1) \stackrel{\text{def}}{=} [\![t_1]\!]\rho = [\![c_1]\!]\rho$ e $P(t \to (t_1, t_2)) \stackrel{\text{def}}{=} [\![t]\!]\rho = [\![(t_1, t_2)]\!]\rho$.

La questione è ancora facilmente risolvibile: infatti

$[\![fst(t)]\!]\rho =$	per definizione
$\mathbf{let}\, v \Leftarrow [\![t]\!]\rho . \pi_1 v =$	per ipotesi induttiva
$\mathbf{let}\, v \Leftarrow [\![(t_1, t_2)]\!]\rho . \pi_1 v =$	per definizione
$\mathbf{let}\, v \Leftarrow \lfloor([\![t_1]\!]\rho, [\![t_2]\!]\rho)\rfloor . \pi_1 v =$	per definizione del let
$\pi_1 \lfloor([\![t_1]\!]\rho, [\![t_2]\!]\rho)\rfloor =$	per definizione di π_1
$[\![t_1]\!]\rho = [\![c_1]\!]\rho$	□

La dimostrazione per la produzione di snd(t) è assolutamente identica.

7.3.5. Applicazione di funzione

Prendiamo la regola dell'applicazione (lazy).

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

Dobbiamo dimostrare

$$P((t_1 \quad t_2) \to c) \stackrel{def}{=} [\![(t_1 \quad t_2)]\!]\rho = [\![c]\!]\rho$$

sotto l'ipotesi di aver già provato $P(t_1 \to \lambda x.t_1') \stackrel{def}{=} [\![t_1]\!]\rho = \left[\!\left[\lambda x.t_1'\right]\!\right]$ e $P(t_1'[^{t_2}/_x] \to c) \stackrel{def}{=} \left[\!\left[t_1'[^{t_2}/_x]\right]\!\right]\rho = [\![c]\!]\rho$

Per farlo useremo il teorema 6.17.

$$\begin{array}{rl}
[\![(t_1 \quad t_2)]\!]\rho = & \text{per definizione} \\
\textbf{let}\,\varphi \Leftarrow [\![t_1]\!]\rho.\varphi([\![t_2]\!]\rho) = & \text{per ipotesi induttiva} \\
\textbf{let}\,\varphi \Leftarrow \left[\!\left[\lambda x.t_1'\right]\!\right]\rho.\varphi([\![t_2]\!]\rho) = & \text{per definizone} \\
\textbf{let}\,\varphi \Leftarrow \lfloor \lambda d.\,[\![t_1]\!]\rho[^d/_x]\rfloor.\varphi([\![t_2]\!]\rho) = & \text{per definizione di let} \\
(\lambda d.\,[\![t_1]\!]\rho[^d/_x])([\![t_2]\!]\rho) & \text{applicando la funzione} \\
[\![t_1]\!]\rho[^{[\![t_2]\!]\rho}/_x] & \text{la semantica denotazionale è composizionale} \\
[\![t_1[^{t_2}/_x]]\!]\rho = & \text{per ipotesi induttiva} \\
[\![c]\!]\rho & \square
\end{array}$$

7.3.6. Ricorsione

Prendiamo la regola della ricorsione

$$\frac{t[^{\textbf{rec}\,x.t}/_x] \to c}{\textbf{rec}\,x.t \to c}$$

Dobbiamo dimostrare

$$P(\textbf{rec}\,x.t \to c) \stackrel{def}{=} [\![\textbf{rec}\,x.t]\!]\rho = [\![c]\!]\rho$$

sotto l'ipotesi di aver già dimostrato $P(t[^{\textbf{rec}\,x.t}/_x] \to c) \stackrel{def}{=} [\![t[^{\textbf{rec}\,x.t}/_x]]\!]\rho = [\![c]\!]\rho$.

Utilizzeremo ancora il teorema 6.17.

$$\begin{array}{rl}
[\![\textbf{rec}\,x.t]\!]\rho = & \text{per la definizione della semantica denotazionale di rec} \\
fix(\lambda d.\,[\![t]\!]\rho[^d/_x]) = & \text{per definizione di punto fisso} \\
(\lambda d.\,[\![t]\!]\rho[^d/_x])(fix(\lambda d.\,[\![t]\!]\rho[^d/_x])) = & \text{applicando la funzione} \\
[\![t]\!]\,[^{fix(\lambda d.[\![t]\!]\rho[^d/_x])}/_x] = & \text{per definizione della semantica denotazionale di rec} \\
[\![t]\!]\rho[^{[\![\textbf{rec}\,x.t]\!]}/_x] = & \text{per il teorema sopra citato} \\
[\![t[^{\textbf{rec}\,x.t}/_x]]\!]\rho = & \text{per ipotesi induttiva} \\
[\![c]\!]\rho & \square
\end{array}$$

7.3.7. Conclusione

Abbiamo dunque dimostrato per casi che vale $\forall t, c.P(t \to c) \stackrel{def}{=} [\![t]\!]\rho = [\![c]\!]\rho$. Questo ci può bastare per affermare che $t \downarrow\Rightarrow t \Downarrow$?

Certamente, come ci dice il teorema seguente.

Teorema 7.2 (Se t converge operazionalmente allora converge anche denotazionalmente)
$t \downarrow\Rightarrow t \Downarrow$

Dimostrazione. É facilissimo vederlo notando che se la semantica operazionale termina, abbiamo che la semantica denotazionale da un risultato uguale alla semantica denotazionale di una forma canonica. Visto che la semantica denotazionale di una forma canonica non da mai $\perp$, questa terminerà sempre. ☠

7.4. La terminazione denotazionale implica la terminazione operazionale

Siamo alla seconda parte della dimostrazione. Vogliamo dimostrare che la terminazione denotazionale implica la terminazione operazionale: il modo naturale di farlo è quello di seguire la procedura utilizzata a suo tempo per IMP, cioé dimostrare il fatto per induzione strutturale.

$$P(t) \stackrel{\text{def}}{=} t \Downarrow \Rightarrow t \to c$$

Questa dimostrazione funziona tranquillamente per quasi tutti i casi, con le modalità di dimostrazione viste precedentemente. Però la dimostrazione non riesce per l'applicazione funzionale.

Proviamo a convincercene. Vogliamo dimostrare

$$P((t_1\ t_2)) \stackrel{\text{def}}{=} (t_1\ t_2) \Downarrow \Rightarrow (t_1\ t_2) \to c$$

Sviluppiamo la semantica denotazionale. Abbiamo che

$$[\![(t_1\ t_2)]\!]\rho \neq \perp \Rightarrow \mathbf{let}\,\varphi \Leftarrow [\![t_1]\!]\rho.\varphi([\![t_2]\!]\rho) \neq \perp$$

Questo significa in primo luogo che $[\![t_1]\!]\rho$ non è $\perp$ e dunque la sua valutazione denotazionale termina. Questo significa ovviamnente che anche la sua valutazione operazionale termina e restituisce una forma canonica: abbiamo visto dunque che $t_1 \to \lambda x.t'$, prima premessa della regola operazionale dell'applicazione. Per il teorema visto nella sezione precedente, questo significa che $[\![t_1]\!]\rho = [\![\lambda x.t']\!] = \lfloor \lambda d.\, [\![t']\!]\rho[^d/_x] \rfloor$.

Sostituendo nel let precedentemente visto questa espressione al posto di φ, abbiamo

$$\left(\lambda d.\, [\![t']\!]\rho[^d/_x]\right)([\![t_2]\!]\rho) = [\![t']\!]\rho\left[^{[\![t_2]\!]\rho}/_x\right] = \left[\!\left[t'\left[^{t_2}/_x\right]\right]\!\right]\rho$$

grazie al teorema 6.17. Siamo dunque arrivati a dover assumere la nostra proprietà su un termine che non è sottotermine di quello originario, cosa che ovviamente non possiamo fare. La nostra unica possibilità é quella di dimostrare questa proprietà con una particolare induzione combinata. La dimostrazione é particolarmente lunga e noiosa, dunque la saltiamo a pié pari.

7.5. Conclusioni

Siamo arrivati dunque a concludere che, se $[\![t_1]\!]\rho = [\![t_2]\!]\rho$ e $t_1 \to c_1$, $t_2 \to c_2$ allora possiamo affermare che $[\![c_1]\!]\rho = [\![c_2]\!]\rho$. Sarebbe bello poter affermare che $c_1 = c_2$, ma questo in generale non è possibile.

L'unico caso nel quale riusciamo a dimostrarlo è il caso in cui il tipo delle due espressioni é int.

Teorema 7.3 (Uguaglianza fra semantiche per termini di tipo *int*)
Se valgono

$$[\![t_1 : \mathbf{int}]\!] = [\![t_2 : \mathbf{int}]\!]$$
$$t_1 \to c_1$$
$$t_2 \to c_2$$

allora vale anche $c_1 = c_2$

Dimostrazione. Sappiamo che sotto le ipotesi sopra citate vale $[\![c_1]\!]\rho = [\![c_2]\!]\rho$. Ma se t_1 e t_2 sono di tipo int la loro forma canonica sarà un valore naturale: abbiamo dunque

$$\begin{array}{l}[\![c_1]\!]\rho = \lfloor n_1 \rfloor \\ [\![c_2]\!]\rho = \lfloor n_2 \rfloor\end{array} \Rightarrow n_1 = n_2$$

Il problema arriva con i tipi composti: è dimostrabile che l'uguaglianza fra le forme canoniche non vale per *nessun* tipo composto (ovvero: esiste sempre un controesempio).

Teorema 7.4 (Divergenza fra semantiche per termini di tipo composto)
Il predicato

$$[\![t_1]\!]\rho = [\![t_2]\!]\rho \Rightarrow (t_1 \uparrow \wedge t_2 \uparrow) \vee (t_1 \to c \wedge t_2 \to c)$$

è vero solo se $t_1 : int$ e $t_2 : int$.

Dimostrazione. Dobbiamo far vedere che per i tipi τ diversi da int la proprietà non vale: visto che è quantificata universalmente dovremo riuscire a trovare un controesempio per ciascun tipo. Il problema é come al solito che i tipi sono infiniti.

Dovremo utilizzare un metodo induttivo. Definiamo per prima cosa una relazione ben fondata sull'insieme dei tipi tale che

- $\tau < \sigma \times \tau$
- $\tau < \sigma \to \tau$

Per questa relazione avremo bisogno di un caso di base, un $\perp$: visto che questo non può essere *int*, per il quale la proprietà vale come abbiamo appena visto, utilizzeremo i due tipi appena superiori rispetto all'ordinamento appena visto. Prenderemo dunque i due generici tipi $\sigma \times int$ e $\sigma \to int$.

L'idea è di

- trovare un controesempio per i casi di base $\sigma \times int$ e $\sigma \to int$
- dato un controesempio per τ, trovare un controesempio per $\sigma \times \tau$ e $\sigma \to \tau$.

Caso base $\sigma \times int$

Prendiamo le due forme

$$(c, 0) \qquad\qquad (c, 0 + 0)$$

Entrambe sono forme canoniche di tipo $\sigma \times int$. La semantica denotazionale di entrambe le forme è $(c, 0)$, in quanto c avrà stessa semantica essendo uguale e la semantica di 0+0 é uguale a quella di 0. Operazionalmente parlando però si riducono a se stesse: la semantica operazionale della coppia é la coppia stessa! Quindi hanno uguale forma denotazionale e diversa semantica operazionale: é il controesempio che cercavamo.

Caso base $\sigma \to int$

Prendiamo similmente le due forme

$$\lambda x.0 \qquad\qquad \lambda x.0 + 0$$

Anche qua denotazionalmente ci riduciamo alla stessa forma, ma operazionalmente si tratta di una forma canonica che non viene semplificata. Ancora la semantica denotazionale è uguale e la semantica operazionale é differente: abbiamo trovato il nostro secondo controesempio.

Abbiamo finito i due casi di base, vediamo adesso i casi induttivi.

Caso induttivo $\sigma \times \tau$ dato un controesempio per τ

Lavoriamo sotto l'ipotesi che per τ abbiamo due termini t_1, t_2 che ci forniscono un controesempio. Allora ovviamente le espressioni (c, t_1) e (c, t_2) saranno denotazionalmente uguali, in quanto la semantica denotazionale di c non cambia e $[\![t_1]\!]\rho = [\![t_2]\!]\rho$, ma saranno operazionalmente diverse essendo le semantiche operazionali diverse a loro volta.

Caso induttivo $\sigma \to \tau$ dato un controesempio per τ

Similmente al caso appena visto, possiamo prendere le forme $\lambda x.t_1$ e $\lambda x.t_2$ se t_1 e t_2 sono controesempi per τ. ☠

Viene da chiedersi perché HOFL ha questo comportamento così diverso da quello di IMP. Il motivo é nel fatto che gestiamo tipi di dato funzionali: é facile dire quando un intero é uguale ad un altro, mentre al contrario il problema di determinare se una funzione é uguale ad un altra é *computazionalmente indecidibile* per il teorema di Rice.[1] Nessuno ci può dunque salvare da questo inconveniente in caso di funzioni di tipo composto, in quanto il problema é impossibile da risolvere.

Un'altra soluzione sarebbe quella di togliere i $\perp$ aggiuntivi per ogni tipo composto, ma in questo caso non potremmo dimostrare l'equivalenza fra le semantiche. La scelta fra l'una o l'altra semantica denotazionale non è assoluta: dipende dall'utilizzo che vogliamo farne. Se abbiamo bisogno dell'equivalenza fra semantiche utilizzeremo la versione con i bottom aggiunti, altrimenti utilizzeremo la versione nella quale non é sicura la terminazione ma quando terminano assieme é sicuro che diano la stessa forma canonica.

[1] Il teorema di Rice afferma che tutti i problemi che hanno come soluzione un insieme di algoritmi che non sia vuoto o non comprenda tutti gli algoritmi é indecidibile: in questo caso l'insieme delle formule con la stessa semantica é sicuramente non vuoto né comprende tutti gli algoritmi!

Parte III.

Calcoli di processo

8. Il linguaggio CCS

TODO-para: citare i seguenti argomenti: Guarded Command Language (GCL) Edsger Dijkstra, Hoare CSP occam

Passiamo alla terza parte del corso, nella quale introduciamo le cosiddette *algebre di processo*, dette anche Calcoli di Processo. Sono teorie per modellare il comportamento, in generale non-deterministico, di processi (o agenti) indipendenti tra loro che possano anche compiere delle computazioni infinite.

Dentro queste algebre di processo esistono molti aspetti di nondeterminismo. Ad esempio, uno di tali aspetti è legato la problema di determinare chi tra due processi terminerà prima (*race-conditions*), mentre altri sono dovuti alla situazione di altri processi nello stesso momento.

Alcuni approcci al problema, come ad esempio il μ-calcolo, rappresentano il tempo esplicitamente nel modello. Con un modello temporizzato potremmo, ad esempio, concludere che un processo arriverà sempre prima di un altro perché più veloce, tuttavia questo non porterebbe grossi vantaggi: elimineremmo solamente una fonte di nondeterminismo. il non determinismo dipende da vari fattori oltre alla velocità di esecuzione ed è insito nel paradigma di programmazione delle algebre di processo che studieremo.

Una cosa che faremo sarà introdurre le computazioni infinte nel nostro paradigma. Tecnicamente erano incluse anche in IMP ed HOFL: programmi come `rec x.x` oppure `while true do skip` non terminano ma avevano una semantica. Tuttavia la loro semantica era sempre pari a $\perp$: non ci interessavano le computazioni infinite, dunque davamo a tutte una semantica che significava errore. Qua invece, come vedremo, le computazioni infinite non solo avranno semantica che le distingue ma saranno anche centrali nel nostro modello di calcolo.

Le due componenti di non-determinismo e computazioni infinite rendono i domini molto più complessi e la riceca di una semantica denotazionale ha portato allo sviluppo di strumenti matematici adeguati nel tempo. Noi daremo piuttosto una *semantica astratta* basata sulla nozione di bisimilarità, secondo la quale due sistemi si equivalgono quando, visti dall'esterno, hanno uno stesso comportamento (cioè hanno lo stesso *comportamento osservabile*.

Vediamo adesso con un esempio come potrebbe essere un possibile programma stile CCS.

Esempio 8.1 (Algebre di processo)
In figura 8.1 possiamo vedere la forma di una cella (che rappresenta un processo) in una possibile algebra di processo. Il calcolo consisterà in una composizione di tali celle.

Ognuna di queste celle ha quattro canali di comunicazione, che possiamo vedere come siti nei quali scambiare informazioni. Possiamo eseguire due possibili azioni su ciascun canale:

- ***inviare** dati: questa azione è indicata con il nome del canale, ad esempio δ, sovrastato da una linea orizzontale e seguito dal dato che vogliamo inviare ($\overline{\delta}(x)$).*

- ***ricevere** dati: questa azione è indicata dal nome del canale seguito dall'identificatore della variabile a cui vogliamo associare il dato ricevuto.*

Diamo un nome ai quattro canali: li chiamiamo semplicemente $\alpha, \beta, \gamma, \delta$. Non è definito a priori quali saranno di input e quali di output: nel nostro esempio utilizzeremo α e δ per l'input e gli altri due per l'output, ma in generale possono anche essere utilizzati in maniera bidirezionale.

La nostra cella può alternativamente trovarsi in 4 possibili stati, ognuno con un numero variabile di parametri. Vediamo questi stati:

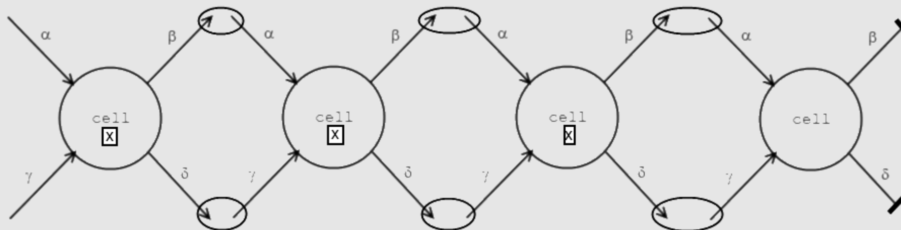
- $CELL_1(y) = \alpha x.CELL_2(x, y) + \bar{\gamma} y.CELL_0$
  In questo stato, chiamato $CELL_1$, abbiamo la presenza di un argomento y. L'operatore $+$ determina

il comportamento non deterministico: può infatti eseguire indifferentemente l'espressione a sinistra o a destra. Nel primo caso riceve in input il valore x sul canale α e si sposta nello stato $CELL_2$ con 2 parametri, nel secondo manda in output, sul canale γ il valore y e si sposta nello stato $CELL_0$ con 0 parametri. è importante notare che questa scelta è completamente nondeterministica: non esiste nessun motivo per cui a priori si possa capire quale strada prenderà l'agente. La decisione verrà presa a runtime, e dipenderà tipicamente da quel che farà chi sarà dall'altra parte del canale o anche semplicemente dal caso.

- $CELL_2(x, y) = \bar{\beta}y.CELL_1(x)$
  Qui il comportamento è deterministico: un agente in questo stato può solamente mandare uno dei suoi vecchi *elementi, in particolare y, sul canale β. Lo stato risultante è $CELL_1$ con il parametro rimanente.*

- $CELL_0 = \delta x.\,\textbf{if}\ x = \$\ \textbf{then}\ ENDCELL\ \textbf{else}\ CELL_1(x)$
  Aspetta un input su δ: se si tratta di un valore qualunque transisce in $CELL_1$ mantenendo tale valore, mentre se invece è il valore speciale $ transisce nello stato ENDCELL, che vedremo tra poco

L'idea è di usare tante celle collegate tra loro a formare uno stack di dimensione variabile nel quale tutte le celle sono nello stato $CELL_1$ e contengono un valore, eccetto l'ultima ENDCELL.



Quando viene aggiunto un nuovo valore la cella testa, a sinistra nel disegno, va in $CELL_2$ e propaga il valore lungo lo stack facendo passare le celle in $CELL_2$ fino a raggiungere ENDCELL che genera una nuova cella e si ritorna nello stato di partenza ma con una cella, e quindi un valore, in più.

Rimane da vedere lo stato ENDCELL, che rappresenta lo stato nel quale una cella attende di essere riempita o si spegne per inattività.

- $ENDCELL = \underbrace{(\alpha x.CELL_1(x) \smile ENDCELL)}_{cresce} + \gamma\$.\underbrace{NIL}_{decresce}$ *è lo stato della cella più a destra: se riceve un valore va nello stato $CELL_1$ e crea una nuova cella nello stato ENDCELL, allungando la sequenza, altrimenti se riceve $ termina la propria esecuzione. Notare che la cella creata può comunicare solo con la sua creatrice: in seguito vedremo qual'è il significato di questi canali privati.*

Un altro importante aspetto della comunicazione in CCS è che rappresenta il *solo* istante di sincronizzione tra i processi, che altrimenti sono completamente asincroni tra loro; essendo le comunicazioni possibili solo tra 2 agenti questo mostra inoltre che esistono esclusivamente sincronizzazioni a due.

Come vedremo, abbiamo utilizzato una versione di CCS non pura. Nel CCS classico non esisterà la comunicazione di valori: i canali ci serviranno solo per la sincronizzazione fra processi.

8.1. Definizione del linguaggio

Definiamo adesso la sintassi del linguaggio CCS, definito da Robin Milner nel 1980. Prima cosa da vedere sono le generiche azioni da compiere: la loro sintassi è così definita.

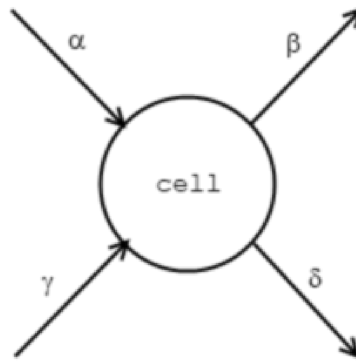


Figura 8.1.: Una cella in un'algebra di processo come CCS

$\Delta ::= \alpha, \beta, \gamma, \delta$	Azioni: generico nome μ
$\Lambda ::= \Delta \mid \overline{\Delta}$	Azioni visibili: generico nome λ
$\tau \notin \Lambda$	Azione invisibile

Come abbiamo visto, i canali di comunicazione hanno nomi α, β, δ e γ. In genere un'azione può essere di input, indicata dal solo nome del canale, o di output, indicata dal nome del canale sovrastato da una riga: per indicare una generica azione visibile utilizzeremo la lettera λ.

Esiste una particolare azione, detta *azione invisibile*. Questa azione è il risultato di una sincronizzazione fra due processi, che non deve essere visibile dall'esterno. Con la semantica operazionale daremo una definizione più chiara. Indicheremo una generica azione, visibile o non visibile, con la lettera greca μ.

8.1.1. Sintassi di CCS

Vediamo la sintassi del linguaggio.

$$p ::= x \quad | \quad nil \quad | \quad \mu.p \quad | \quad p\backslash\alpha \quad | \quad p[\Phi] \quad | \quad p+p \quad | \quad p \mid p \quad | \quad \mathbf{rec}\, x.p$$

Cerchiamo innanzitutto di vedere cosa questi simboli significhino ad alto livello, per dare una semantica precisa a breve.

x è un generico nome. Lo utilizzeremo principalmente per la ricorsione.

nil è il processo vuoto o terminato

μ**.p** è un prefisso, composto da un'azione e da un processo

p $\backslash\alpha$ è una restrizione: p non può eseguire mosse sul canale α. Notare che le mosse di sincronizzazione saranno indicate con τ e dunque saranno permesse da questo costrutto. Vedremo a breve la semantica operazionale che ci chiarirà le idee.

p[Φ] sostituzione di nomi all'interno di p

p + p costrutto per il nondeterminismo: con questo costrutto si indica che p può seguire indifferentemente l'una o l'altra strada.

p|p costrutto parallelo: i due processi sono eseguiti "contemporaneamente" (in realtà faranno un passo ciascuno, il parallelismo è solo concettuale)

recx.p Costrutto ricorsivo. Sostituisce se stesso in tutte le occorrenze della variabile x in p.

Gli agenti CCS sono termini chiusi.

8.1.2. Semantica Operazionale

La semantica operazionale del CCS è definita da un sistema di transizioni etichettato (LCS - *Labeled Transition System*), i cui nodi sono stati del calcolo (a tutti gli effetti processi CCS) e i cui archi sono etichettati con azioni, che siano τ o λ. Le regole di transizione partiranno da un processo e cercheranno di determinare gli archi uscenti da tale processo.

Una peculiarità di CCS è quello di avere un unico sistema di transizione per tutti i processi che compongono il sistema: questo ci permetterà di modellare in maniera molto più precisa le sincronizzazioni e le interazioni fra i processi.

Le formule ben formate del nostro sistema saranno triple composte da uno stato, una transizione ed uno stato di arrivo. Il formato sarà $p_1 \xrightarrow{\mu} p_2$ ed indicherà che il processo p_1 sarà capace di eseguire un'azione μ e diventare p_2.

$$\text{(Act)} \quad \frac{}{\mu.p \xrightarrow{\mu} p}$$

L'unico assioma della semantica è quello che regola il prefisso: transendo con l'azione si va nel processo p.

$$\text{(Res)} \quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha} \; \mu \neq \alpha, \bar{\alpha}$$

Se abbiamo una restrizione, dobbiamo semplicemente evitare di compiere azioni che coinvolgano la sua variabile. Per il resto transiamo normalmente.

$$\text{(Rel)} \quad \frac{p \xrightarrow{\mu} q}{p[\Phi] \xrightarrow{\Phi(\mu)} q[\Phi]}$$

Questa regola può sembrare complessa, ma ha significato molto semplice: se p transisce con μ in q, $p[\Phi]$ transisce con $\Phi(\mu)$ in $q[\Phi]$. La sostituzione Φ è una semplice ridenominazione dei canali.

$$\text{(Sum)} \quad \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$$

Ecco la regola che muove il nondeterminismo. Semplicemente possiamo scegliere uno qualunque dei due rami *dimenticando completamente* l'altro: tutte le informazioni contenute nell'altro lato sono perse.

$$\text{(Com)} \quad \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

Il parallelo può invece muovere l'uno o l'altro processo separatamente, portandosi dietro la controparte. Il parallelismo è dunque un'esecuzione a turni dei due processi.

$$\text{(Sync)} \quad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\bar{\lambda}} q_2}{p_1|q_1 \xrightarrow{\tau} p_2|q_2}$$

Ecco la regola della sincronizzazione. Il primo processo esegue una generica mossa λ, mentre l'altro processo esegue una mossa sullo stesso canale e di segno contrario. I due processi a questo punto si sincronizzano: la loro composizione parallela esegue una mossa τ.

Notare che un processo composto da due processi paralleli ristretto su un canale λ può eseguire la sincronizzazione su λ in quanto questa ha come risultato una mossa τ.

$$(\text{Rec}) \quad \frac{p[\textbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\textbf{rec}\, x.p \xrightarrow{\mu} q}$$

Il comando ricorsivo è esattamente analogo a quello di HOFL: a tutte le occorrenze del nome x sostituiamo il testo completo della ricorsione.

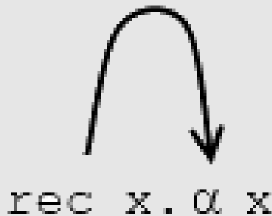
Esempio 8.2 (Esempio di derivazione)
Prendiamo il processo CCS così definito.

$$(((\textbf{rec}\, x.\alpha x + \beta x)|(\textbf{rec}\, x.\alpha x + \gamma x))|\,\textbf{rec}\, x.\bar{\alpha}x)\backslash\alpha$$

Proviamo innanzitutto a capire come si muoverebbe il terzo processo parallelo se fosse solo e non esistesse la restrizione su α.

$$\textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\mu} q \longleftarrow \bar{\alpha}\,\textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\mu} q \xleftarrow{\mu=\bar{\alpha} \quad q=\textbf{rec}\, x.\bar{\alpha}x} \square$$

Nella prima mossa sviluppiamo il **rec** $x.\alpha x$ *riscrivendo se stesso all'interno del processo. Dunque mediante l'assioma muoviamo su α ritornando nello stato iniziale. Possiamo esprimere questa situazione con un grafo così fatto*



In pratica questo agente continua a ciclare sulla stessa azione all'infinito. Tuttavia, questa azione non è possibile per l'agente completo: la restrizione su α ci impedisce di muovere un processo singolo su tale mossa. Per muovere su α il terzo processo dovrà trovare qualcuno che si sincronizzi su tale canale, in modo che all'esterno si veda una mossa τ.

Nel processo completo dunque dovremo muoverci per sincronizzazioni: il terzo processo funzionerà come una sorta di arbitro che deciderà quale degli altri due processi muovere.

$$(((\textbf{rec}\, x.\alpha x + \beta x)|(\textbf{rec}\, x.\alpha x. + \gamma x))|\,\textbf{rec}\, \bar{\alpha}x)\backslash\alpha \xrightarrow{\mu} q \xleftarrow{q=q'\backslash\alpha}$$
$$((\textbf{rec}\, x.\alpha x + \beta x)|(\textbf{rec}\, x.\alpha x + \gamma x))|\,\textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\mu} q' \xleftarrow{\mu=\tau \quad q'=q_1|q_2 \quad \mu\neq\lambda,\bar{\alpha}}$$
$$(\textbf{rec}\, x.\alpha x + \beta x)|\,\textbf{rec}\, x.\alpha x + \gamma x \xrightarrow{\alpha} q_1 \quad \textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\bar{\alpha}} q_2 \xleftarrow{q_1=(\textbf{rec}\, x.\alpha x+\beta x)\;|\;q_3}$$
$$\textbf{rec}\, x.\alpha x + \gamma x \xrightarrow{\alpha} q_3 \quad \textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\bar{\alpha}} q_2 \longleftarrow$$
$$\alpha(\textbf{rec}\, x.\alpha x + \gamma x) + \gamma(\textbf{rec}\, x.\alpha x + \gamma x) \xrightarrow{\alpha} q_3 \quad \textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\bar{\alpha}} q_2 \longleftarrow$$
$$\alpha(\textbf{rec}\, x.\alpha x + \gamma x) \xrightarrow{\alpha} q_3 \quad \textbf{rec}\, x.\bar{\alpha}x \xrightarrow{\bar{\alpha}} q_2 \xleftarrow{q_3=\textbf{rec}\, x.\alpha x+\gamma x}$$
$$\bar{\alpha}(\textbf{rec}\, x.\bar{\alpha}x) \xrightarrow{\bar{\alpha}} q_2 \xleftarrow{q_2=\textbf{rec}\, x.\bar{\alpha}x}$$
$$\square$$

$$q_2 = \textbf{rec}\, x.\bar{\alpha}x$$
$$q_3 = \textbf{rec}\, x.\alpha x + \gamma x$$
$$q_1 = (\textbf{rec}\, x.\alpha x + \beta x)|q_3 = (\textbf{rec}\, x.\alpha x + \beta x)|\,\textbf{rec}\, x.\alpha x + \gamma x$$
$$q' = q_1|q_2 = ((\textbf{rec}\, x.\alpha x + \beta x)|(\textbf{rec}\, x.\alpha x + \gamma x))|\,\textbf{rec}\, x.\bar{\alpha}x$$
$$q = q'\backslash\alpha = (((\textbf{rec}\, x.\alpha x + \beta x)|(\textbf{rec}\, x.\alpha x + \gamma x))|\,\textbf{rec}\, x.\bar{\alpha}x)\backslash\alpha$$
$$\mu = \tau$$

NOTA-paracetamolo: l'esempio fatto a lezione è diverso perché alla terza riga a fatto una scelta diversa credo, e poi usa sempre λ nelle transizioni invece che α: CONTROLLARE!!!

Definizione 8.3 (Collegamento di due agenti)
L'operazione di collegamento di due agenti, consiste nel rinominare i canali da connettere e renderli non visibili dall'esterno:

TODO aggiungere immaginette

$$p \circledcirc q = (P[^{\vartheta}/_{\beta},^{\eta}/_{\delta}] \mid Q[^{\vartheta}/_{\alpha},^{\eta}/_{\gamma}])\backslash\vartheta\backslash\eta$$

I nuovi canali η e ϑ vengono anche detti α-convertibili, locali o segreti.

8.1.3. Semantica astratta del CCS

D'ora in poi lavoreremo solo con particolare agenti detti *guarded*, per motivi che illustreremo in seguito.

Definizione 8.4 (Agenti Guarded)
Un agente è detto guarded *se nell'operatore* **rec** *$x.p$ ogni occorrenza libera di x in p è in un sottotermine di p del tipo $\mu.q$.*

Intuitivamente ogni occorrenza di x deve essere preceduta da un'azione μ, altrimenti si ottengono infinite azioni identiche fra loro e questo, in seguito, darà problemi di continuità nella funzione Φ.

Lemma 8.5
Se p è un agente guarded allora $\{q \mid p \xrightarrow{\mu} q\}$ è finito.

Esempio 8.6 (Agente non guardato)
L'agente può fare solo 2 mosse, svolgiamole entrambe

$$\begin{aligned}
&\textbf{rec}\, x.x|\alpha\,\textbf{nil} \xrightarrow{\mu}\longleftarrow\\
&(\textbf{rec}\, x.x|\alpha\,\textbf{nil})|\alpha\,\textbf{nil} \xrightarrow{\mu}\longleftarrow\\
&(\textbf{rec}\, x.x|\alpha\,\textbf{nil})|\,\textbf{nil}\,|\alpha\,\textbf{nil} \xrightarrow{\mu}\longleftarrow
\end{aligned}$$

in generale può fare N mosse

$$(\textbf{rec}\, x.x|\alpha\,\textbf{nil})|\,\textbf{nil}\,|\underbrace{\alpha\,\textbf{nil}\,|\alpha\,\textbf{nil}\,|\ldots}_{N}$$

La semantica del CCS può essere vista *sistema di transizioni etichettato*, di cui un particolare agente forma un sottoinsieme, gli stati che può raggiungere. In generale questo sottoinsieme può essere infinito dato che il CCS è Turing–equivalente, tuttavia nel caso sia finito (anche se vastissimo) è possibile stabilirne alcune proprietà (*model checking*).

8.2. Relazioni di equivalenza

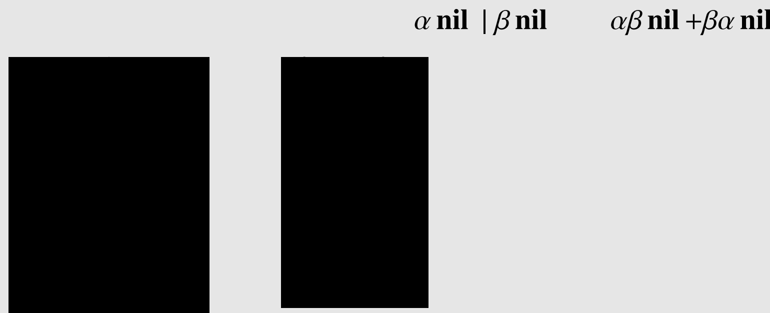
La semantica che abbiamo visto ora per CCS è molto più particolareggiata di quelle già studiate per IMP o HOFL, cerchiamo allora un metodo di astrazione per poter stabilire equivalenze tra agenti.

8.2.1. Isomorfismo dei grafi

Come primo passo di astrazione scegliamo di trascurare i nomi dei nodi, e consideriamo solo le transizioni: 2 agenti isomorfi sono considerati equivalenti.

Esempio 8.7 (Agenti isomorfi)

$$\alpha\,\mathbf{nil}\ |\ \beta\,\mathbf{nil} \qquad \alpha\beta\,\mathbf{nil} + \beta\alpha\,\mathbf{nil}$$



I due agenti sono isomorfi, fanno le stesse mosse.

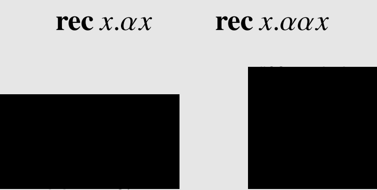
NOTE: Abbiamo fornito una semantica interleaving: è possibile osservare solo una mossa per volta.

8.2.2. Stringhe o *Trace equivalence*

Dato che rappresentiamo i nostri agenti con sistemi di transizioni etichettati (possibilemente con stati infiniti), la teoria degli automi ci suggerisce che potremmo basare l'equivalenza sull'insieme delle stringhe riconosciute.

Esempio 8.8 (Grafi differenti, stringhe uguali, comportamento uguale)
Prendiamo 2 agenti

$$\mathbf{rec}\ x.\alpha x \qquad \mathbf{rec}\ x.\alpha\alpha x$$



Il primo fa una mossa α e ritorna nello stesso stato.

$$\mathbf{rec}\ x.\alpha\alpha x \xrightarrow{\mu} q \longleftarrow \alpha\alpha\,\mathbf{rec}\ x.\alpha\alpha x \xrightarrow{\mu} q \longleftarrow \ldots$$

*Il secondo fa una mossa α va in uno stato intermedio α **rec** $x.\alpha\alpha x$ e ritorna allo stato iniziale.*

In questo caso i grafi non sono isomorfi e quindi il nostro principio considera i due automi differenti anche se in realtà questi hanno un comportamento praticamente identico. Infatti se vediamo il loro sviluppo

(*unfolding*) entrambi generano le stesse stringhe di α, solo che uno le "arrotola" in due passi, è facile trovare altri infiniti agenti che svolgano la stessa cosa in 3,4,... passi.

Esempio 8.9

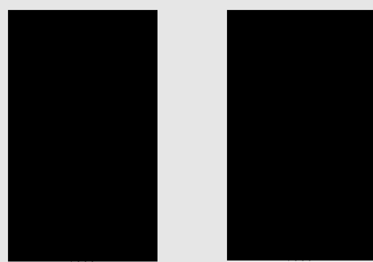
$$\alpha\,\mathbf{nil} + \alpha\,\mathbf{nil} \qquad \alpha\,\mathbf{nil}$$

Il grafo è identico.

Tuttavia esistono casi in cui né le stringhe generate né l'isomorfismo ci aiutano a caratterizzare il comportamento di un automa.

Esempio 8.10 (Grafi differenti, stringhe uguali, comportamento differente)
Vediamo ora un esempio interessante

$$\alpha(\beta\,\mathbf{nil} + \gamma\,\mathbf{nil}) \qquad \alpha\beta\,\mathbf{nil} + \alpha\gamma\,\mathbf{nil}$$



I grafi non sono isomorfi, ma se andiamo a guardare le stringhe di azioni, gli insiemi sono uguali, ossia $\{\alpha\beta, \alpha\gamma\}$
Tuttavia logicamente hanno un comportamento diverso perché il primo decide se fare β o γ dopo aver fatto α il secondo invece deve scegliere dall'inizio quale dei due α fare se poi vuole fare β o γ.

Da questi esempi vediamo come sia l'uguaglianza a stringhe, sia l'omomorfismo dei grafi non rispecchino la nostra idea intuitiva di due agenti equivalenti.

8.3. Bisimilarità

Vediamo ora la relazione detta *Bisimilarità* che dimostreremo corrispondere al nostro concetto intuitivo di uguaglianza di agenti e vedremo avere una serie di buone proprietà.
Innanzitutto cerchiamo una relazione che sia di equivalenza e che quindi rispetti le tre seguenti proprietà.

Definizione 8.11 (Relazione di equivalenza)
Una relazione di equivalenza è una relazione con le seguenti proprietà:

$$\begin{array}{rl} x \equiv x & \textit{riflessività} \\ x \equiv y \wedge y \equiv z \Rightarrow x \equiv z & \textit{transitività} \\ x \equiv y \Rightarrow y \equiv x & \textit{simmetria} \end{array}$$

e permette di suddividere un insieme in sottoinsiemi disgiunti detti classi di equivalenza *all'interno delle quali ogni elemento è equivalente.*

Inoltre cerchiamo una relazione che ci dica quando due agenti sono equivalenti, quindi una relazione su agenti $P \times P$.

Definiamo inoltre il concetto di contesto e congruenza:

Definizione 8.12 (Contesto)
Un contesto è un'espressione algebrica con un 'buco', che indicheremo con $C[\ ]$

Un esempio di contesto è $b + [\ ]$ dove al posto di $[\ ]$ possiamo sostituire una qualsiasi espressione della nostra algebra.

Definizione 8.13 (Congruenza)
Una relazione $\sim_{\mathscr{C}}$ *è una* congruenza *se:* $\forall C[\ ]$

$$P \sim_{\mathscr{C}} Q \quad \Rightarrow \quad C[P] \sim_{\mathscr{C}} C[Q]$$

Diamo un esempio tramite la teoria dei giochi di come deve comportarsi la nostra relazione.

Immaginiamo di avere due agenti, p e q, e due giocatori, Alice e Bob. Lo scopo di Alice è dimostrare che p e q sono differenti e possiede una algoritmo che le permette di fare la mossa migliore per raggiungere il suo scopo. Bob viceversa vuole mostrare che i due agenti sono uguali.
Alice gioca per prima facendo una mossa μ con uno qualunque dei due agenti, poniamo p, $p \xrightarrow{\mu} p'$ e Bob deve rispondere facendo la stessa mossa con l'altro agente q, $q \xrightarrow{\mu} q'$.

Esempio 8.14 (Teoria dei giochi per l'esempio 8.10)
Nell'esempio precedente Alice sceglierebbe α e Bob sarebbe costretto a scegliere una particolare α, poniamo quella a sinistra, a questo punto Alice sapendo che Bob potrebbe solo fare β sceglierebbe γ. Il discorso è speculare se Bob scegliesse di passare a destra e quindi concludiamo che i due agenti sono differenti.

Definiamo più formalmente quanto espresso dall'esempio di Alice e Bob:

Definizione 8.15 (Strong bisimulation)
Una relazione binaria R sull'insieme di stati di un agente è una bisimulazione se

$$\forall s_1 \; R \; s_2 \quad \begin{array}{llll} se \; s_1 \xrightarrow{\alpha} s_1' & allora \; esiste \; una \; transizione & s_2 \xrightarrow{\alpha} s_2' \; tale \; che \; s_1' \; R \; s_2' \\ se \; s_2 \xrightarrow{\alpha} s_2' & allora \; esiste \; una \; transizione & s_1 \xrightarrow{\alpha} s_1' \; tale \; che \; s_1' \; R \; s_2' \end{array}$$

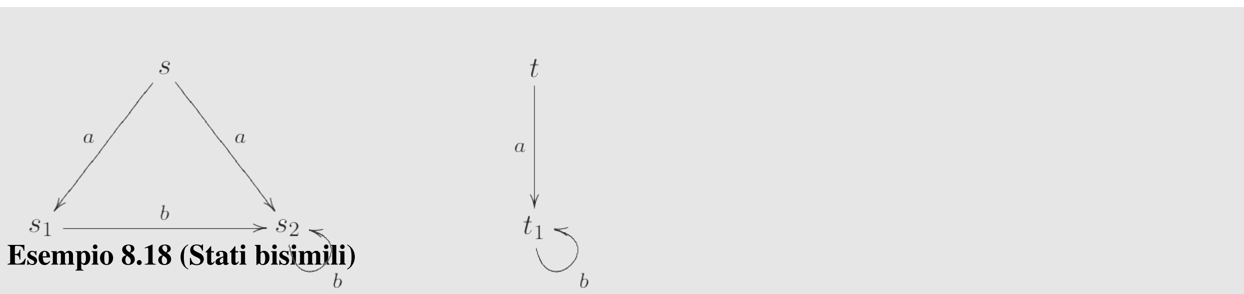
Definizione 8.16 (Strong bisimilarity $\simeq$)
Due stati s e s' sono bisimilari, scritto $s \simeq s'$, se e solo se sono legati da una bisimulazione forte.
La relazione $\simeq$ è detta bisimilarità forte ed è l'unione di tutte le bisimulazioni

$$\simeq \stackrel{\text{def}}{=} \bigcup_{R \sqsubseteq \Phi(R)} R$$

Visto che rapprensentiamo i nostri processi CCS con dei sistemi di transizione etichettati (LTS) i cui stati sono a loro volta processi CCS, questa nozione di bisimulazione si applica tanto ai processi quanto ai loro stati.
Inoltre va notato che la bisimulazione è un concetto generale legato solo ai LTS e quindi di conseguenza al CCS.

Infine si può dimostrare che $\simeq$ rispetta le proprietà di riflessività, simmetria e transitività.

Teorema 8.17
La relazione $\simeq$ di bisimilarità è una relazione di equivalenza.



Esempio 8.18 (Stati bisimili)

Vogliamo mostrare che $s \simeq t$, per farlo dobbiamo trovare una bisimulazione (forte) R tale che $(s, t) \in R$. Definiamo allora R come

$$R = \{(s, t), (s_1, t_1), (s_2, t_1)\}$$

Ovviamente abbiamo che $(s, t) \in R$, ora rimane da dimostrare che R sia effettivamente una bisimulazione. Per farlo per ogni coppia di stati di R dobbiamo analizzare ogni possibile transizione di uno stato e verificare se possono corrispondere a transizioni dell'altro stato, naturalmente le due transizioni devo avere la stessa etichetta.
Questo può essere verificato per ispezione.

Alternativamente se volessimo provare che $s_1 \simeq s_2$ potremmo usare la seguente

$$R = \{(s_1, s_2), (s_2, s_2)\}$$

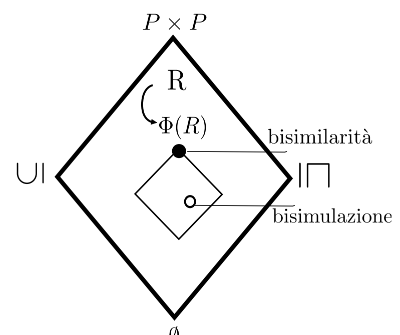
e verificare ancora che si tratta di una bisimulazione.

8.3.1. Punto fisso

La relazione di bisimilarità può essere espressa anche come calcolo di punto fisso di un operatore, il che ci fornisce un metodo più operativo per calcolarla effettivamente.

8.3.1.1. Il $CPO_\perp$ dei processi

Definiamo quindi $(\mathscr{P}(P \times P), \subseteq)$ come il $CPO_\perp$ sul quale lavoriamo e nel quale rappresentiamo le relazioni come insiemi di coppie. Abbiamo in particolare che



$P \times P$ rappresenta la relazione che considera tutti i processi equivalenti e quindi non distingue nulla,

- $\emptyset$ è la relazione per cui un processo non è uguale a nessun altro, addirittura nemmeno a se stesso (in realtà sarebbe puù corretto usare gli infiniti insiemi di coppie del tipo $\{(a,a)\}, \{(b,b)\}, \ldots$).

Tuttavia in questo caso è conveniente considerare il $CPO_\perp$ al "contrario", cioè useremo la relazione $\sqsubseteq = \supseteq$. In questo modo partiremo da $P \times P$ come bottom ed andremo via via raffinando.

8.3.1.2. L'operatore Φ

L'operazione di cui vogliamo fare il punto fisso è Φ che se applicata ad un insieme R, cioè una relazione con una certa capacità di discernere tra agenti, ne ritorna una migliore, più selettiva.

$$p\ \Phi(R)\ q \stackrel{\text{def}}{=} \begin{array}{l} p \xrightarrow{\mu} p' \Longrightarrow \exists q'.\ q \xrightarrow{\mu} q' \quad \text{and} \quad p'Rq' \quad \wedge \\ q \xrightarrow{\mu} q' \Longrightarrow \exists p'.\ p \xrightarrow{\mu} p' \quad \text{and} \quad p'Rq' \end{array}$$

Come è facile vedere questa definizione si sposa perfettamente con la definizione di bisimulazione 8.15 e infatti abbiamo che le bisimulazioni sono proprio i punti fissi di questo operatore.

Teorema 8.19 (Bisimulazione come punto fisso)

$$B = \Phi(B) \quad \textit{dove } B \textit{ è una bisimulazione}$$

Inoltre la bisimilarità può essere ottenuta come massimo punto fisso dell'operatore Φ partendo dal punto $P \times P$ e via via raffinado.
Notare che il limite nel powerset veniva fatto con l'unione, nel nostro dominio "rovesciato" va fatto con l'intersezione.

Teorema 8.20 (Bisimilarità come massimo punto fisso)

$$\simeq = \bigcap_{n \in \omega} \Phi^n(P \times P)$$

Esempio 8.21 (Bisimilarità come massimo punto fisso)
Completiamo l'esempio 8.18 calcolando la bisimilarità come calcolo di un punto fisso

$$\begin{aligned} R_0 &= \{\{s, t, s_1, s_2, t_1\}\} \\ R_1 &= \{\{s, t\}, \{s_1 s_2, t_1\}\} \\ R_2 &= \{\{s, t\}, \{s_1 s_2, t_1\}\} = fix \end{aligned}$$

Abbiamo ottenuto le classi di equivalenza della nostra relazione di equivalenza.
All'interno di ogni classe ogni coppia di elementi è quindi in relazione di equivalenza

$$\simeq = \{(s,t), (s_1, s_2), (s_1, t_1), (s_2, t_1)\}$$

è facile vedere come le due bisimulazioni calcolare precedentemente siano comprese nella nostra bisimilarità, che infatti è l'unione di tutte le possibile bisimulazioni.

Ricapitolando quanto visto:

- le bisimulazioni sono punti fissi dell'operatore $\varPhi$

$$R \sqsubseteq \varPhi(R)$$

- la bisimilarità è massimo punto fisso dell'operatore $\varPhi$ e può essere ottenuta
  - come unione di tutti i punti fissi di $\varPhi$ in accordo al teorema di Tarski

$$\simeq \stackrel{\text{def}}{=} \bigcup_{R \sqsubseteq \varPhi(R)} R$$

  - come calcolo di massimo punto fisso di
    varPhi in accordo al teorema di Kleene

$$\simeq = \bigcap_{n \in \omega} \varPhi^n(P \times P)$$

$$R \sqsubseteq \varPhi(R) \Longleftrightarrow R \text{ è una bisimulazione}$$

cioè R deve essere un pre-punto fisso.

Perché ci occupiamo anche delle bisimulazioni se abbiamo un modo per calcolare direttamente la bisimilarità, che le contiene tutte ed è la più generale possibile?
La ragione è che le bisimulazioni sono più semplici da costruire mentre la bisimilarità, non è nemmeno ricorsivamente numerabile dato che va calcolata per tutti gli stati raggiungibili (che ricodiamo possono essere infiniti).
Notare che se il sottoinsime degli stati raggiungibili è finito TODOnon mi ricordo ;(TODO.

Naturalmente per poter calcolare punti fissi abbiamo bisogno di sapere se $\varPhi$ è monotonia continua.

Teorema 8.22 (Monotonia di Φ)

$$R_1 \sqsubseteq R_2 \Rightarrow \varPhi(R_1) \sqsubseteq \varPhi(R_2)$$

Dimostrazione. Se R_1 è più permissiva di R_2, cioè $p'R_1q' \Rightarrow p'R_2q'$ allora $p\varPhi(R_1)q \Rightarrow p\varPhi(R_2)q$.

Lemma 8.23
Se p è un agente guarded allora $q \mid p \xrightarrow{\mu} q$ *è finito.*

Teorema 8.24
Continuità:

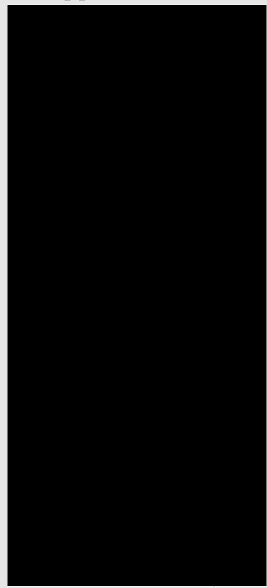
$$\bigcap_{i \in \omega} \varPhi(R_i) = \varPhi(\bigcap_{i \in \omega} R_i)$$

Il fatto che l'operatore sia continuo non è ovvio e noi non lo dimostriamo. Comunuque la proprietà vale se il nostro sistema di transizione è *finite branching* e per questo abbiamo posto la condizione sugli agenti *guarded* 8.4.

Esempio 8.25 (Calcolo Bisimilarità)
Rivediamo l'esempio 8.10 tramite la bisimilarità:

$$p = \alpha(\beta\,\mathbf{nil} + \gamma\,\mathbf{nil}) \qquad q = \alpha\beta\,\mathbf{nil} + \alpha\gamma\,\mathbf{nil}$$

Rappresentiamo tutti gli stati raggiungibili dai due automi.



All'inizio della prova tutti gli elementi sono equivalenti.

$$R_0 = \{\{p, p', \mathbf{nil}, q, q', q''\}\}$$

Applichiamo ϑ:

$$R_1 = \vartheta(R_0) = \{\{p, q\}, \{p'\}, \{q'\}, \{q''\}, \{\mathbf{nil}\}\}$$

$$R_2 = \vartheta(R_1) = \{\{p\}, \{q\}, \{p'\}, \{q'\}, \{q''\}, \{\mathbf{nil}\}\}$$

Questa è una catena discendente nell'insieme, che raffina sempre di più (è monotona).
Ne concludiamo che non ci sono stati equivalenti, proprio come nella nostra intuizione iniziale dei due atomi.

Esempio 8.26 (Punto fisso alla prima iterazione)

$$p = \mathbf{rec}\, x.\alpha\alpha x \qquad q = \mathbf{rec}\, x.\alpha x \qquad p' = \alpha\,\mathbf{rec}\, x.\alpha\alpha x$$

$$R_0 = \{\{p, q, p'\}\}$$
$$R_1 = R_0$$

Sono equivalenti perché tutti fanno solo α.

8.4. Congruenza e bisimilarità

Una proprietà fondamentale dei nostri sistemi è la *composizionalità*, cioè la possibilità di comporre agenti semplici in agenti più complessi o di sostuire parti di un agente.
In particolare per poter sostituire un agente con un altro, è necessario non soltanto che questi siano bisimili

$$P \simeq Q \Longrightarrow C[P] \simeq C[Q]$$

ma che la bisimilarità risulti una congruenza rispetto alle operazioni del linguaggio.

Esempio 8.27 (Equivalenza a stringhe non è congruenza)
Prendiamo un contesto con un "buco"

$$C[_] = (_ \mid \bar{\alpha}\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta$$

sostituiamo ora due agenti che risultavano equivalenti a stringhe:

$$C[p] = ((\alpha \mid \beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \mid \bar{\alpha}\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta$$

$$C[q] = ((\alpha\beta.\,\mathbf{nil} + \alpha\gamma.\,\mathbf{nil}) \mid \bar{\alpha}\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta$$

inseriti in un contesto, due agenti equivalenti a stringhe hanno un comportamento diverso, infatti C[p] si può bloccare mentre C[q] no, questa è infatti la conferma che l'equivalenza a stringhe non è una congruenza.

Dimostriamo quindi che la bisimilarità è una congruenza:

$$x \equiv y \Longrightarrow F(X) \equiv F(Y)$$

Questa proprietà nella semantica denotazionale era garantita per costruzione, avevamo infatti la proprietà di *composizionalità*.

$$[\![t_1]\!]\, p = [\![t_2]\!]\rho \Longrightarrow \left[\!\!\left[t[^{t_1}/_x]\right]\!\!\right]\rho = \left[\!\!\left[t[^{t_2}/_x]\right]\!\!\right]\rho$$

Dobbiamo quindi mostrare per tutte le regole del nostro linguaggio che la bisimilarità è una congruenza, noi mostreremo in dettaglio solo la regola più complessa: il parallelo.

8.4.0.3. Bisimilarità è una congruenza per il parallelo

Vogliamo dimostrare che:

$$p_1 \simeq q_1 \wedge p_2 \simeq q_2 \quad \overset{?}{\Longrightarrow} \quad p_1 \mid p_2 \sim q_1 \mid q_2$$

Abbiamo sicuramente le seguenti bisimulazioni:

$$\begin{array}{ll} p_1\ R_1\ q_1 & R_1 \text{ bisimulazione} \\ p_2\ R_2\ q_2 & R_2 \text{ bisimulazione} \end{array}$$

Per dimostrare la conseguenza ci basta trovare un bisimulazione in cui la proprietà è verificata: costruiamone una ad-hoc.

$$R = \{(p_1|q_1, p_2|q_2)\}\ t.c.\ p_1\ R_1\ p_2\ \wedge\ q_1\ R_2\ q_2\}$$

In questa relazione la nostra proprietà è sicuramente valida per costruzione, rimane da dimostrare che sia davvero una bisimulazione.

$$P(p_1|q_1 \xrightarrow{\mu} p_1'|q_1') \overset{\text{def}}{=} p_1|q_1\ R\ p_2|q_2 \Longrightarrow \exists p_2'.p_2|q_2 \xrightarrow{\mu} p_2'|q_2'\ \wedge\ p_1'|q_1'\ R\ p_2'|q_2'$$

la proprietà si dimostra per induzione sulle regole, cioè per le tre regole del parallelo.

Prima regola

$$\frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q}$$

Per questa regola possiamo muovere uno solo dei due agenti alla volta

$$P(p_1|q \xrightarrow{\mu} p_1'|q) \stackrel{\text{def}}{=} p_1|q \; R \; p_2|q \Longrightarrow \exists p_2'. \; p_2|q \xrightarrow{\mu} p_2'|q \;\wedge\; p_1'|q \; R \; p_2'|q$$

Dalle premesse della regola otteniamo che

$$p_1 \xrightarrow{\mu} p_1'$$

e per ipotesi induttiva abbiamo che

$$P(p_1 \xrightarrow{\mu} p_1') \stackrel{\text{def}}{=} p_1 \; R \; p_2 \Longrightarrow \exists p_2'. \; p_2 \xrightarrow{\mu} p_2' \;\wedge\; p_1' \; R \; p_2'$$

Da cui, se assumiamo la premessa della proprietà, otteniamo che questo p_2' effettivamente esiste e possiamo usarlo nella regola del parallelo, ottenendo

$$p_2|q \xrightarrow{\mu} p_2'|q \qquad \text{prima conseguenza}$$

mentre per come abbiamo costruito la relazione, vale ancora

$$p_1'|q \; R \; p_2'|q \qquad \text{seconda conseguenza}$$

come volevamo dimostrare.
La dimostrazione per la seconda regola del parallelo è simmetrica alla precedente, vediamo la terza.

Terza regola

$$\frac{p_1 \xrightarrow{\lambda} p_1' \quad q_1 \xrightarrow{\bar{\lambda}} q_1'}{p_1|q_1 \xrightarrow{\tau} p_1'|q_1'}$$

La proprietà da dimostrare è

$$P(p_1|q_1 \xrightarrow{\tau} p_1'|q_1') \stackrel{\text{def}}{=} p_1|q_1 \; R \; p_2|q_2 \Longrightarrow \exists p_2', q_2'. \; p_2|q_2 \xrightarrow{\tau} p_2'|q_2' \wedge p_1'|q_1' \; R \; p_2'|q_2'$$

Assumiamo la premessa, e sfruttiamo l'ipotesi induttiva

$$\begin{array}{lll} p_1 \; R_1 \; p_2 & p_2 \xrightarrow{\lambda} p_2' & p_1' \; R_1 \; p_2' \\ q_1 \; R_2 \; q_2 & q_2 \xrightarrow{\bar{\lambda}} q_2' & q_1' \; R_2 \; q_2' \end{array}$$

Da cui usando le prime conseguenze, otteniamo dalla regola del parallelo

$$p_2|q_2 \xrightarrow{\tau} p_2'|q_2'$$

e dalle seconde conseguenza sappiamo che vale ancora

$$p_1'|q_1' \; R \; p_2'|q_2'$$

come volevamo dimostrare.

8.4.0.4. Accenno di dimostrazione di bisimilarità per regola del prefisso

Per ipotesi abbiamo una bisimilarità R, ne dobbiamo trovare un'altra R_1

$$pRq \qquad R' = R \cup \{(\mu.p, \mu.q)\}$$

quindi abbiamo la stessa relazione di prima con in più proprio la coppia di cui avevamo bisogno.

Si potrebbe pensare che la bisimilarità sia una relazione di equivalenza arbitraria, e che in alcuni casi possa distinguere troppo o troppo poco, introduciamo un formalismo che ci permetterà di dimostrare che è la corretta nozione di equivalenza.

8.5. Logica di Hennessy - Milner

Nel 1980 Hennessy e Milner introdussero una logica modale (cioè una logica che permette i concetti di *possibile* o *necessario*) usata per dimostrare proprietà di un sistema di transizione etichettato (e quindi sulla semantica operazionale del CCS).

$$F ::= \neg F \quad | \quad \wedge_{\forall i \in I} F_i \quad | \quad \diamond_\mu F$$

Il primo è il noto operatore di negazione, il secondo è un AND di vari F, per il quale vale in particolare $\wedge_{i \in I} F_i = true$.
Il terzo è un *operatore modale*: se esiste almeno una transizione etichettata μ nel cui stato di arrivo vale F allora nello stato di partenza vale $\diamond_\mu F$.

Definizione 8.28 (Satisfaction relation)
La satisfaction relation *è una relazione che fornisce tutte le formule soddisfatte da un certo agente*

$$\models \subseteq P \times \mathscr{L}$$

con P gli agenti e $\mathscr{L}$ l'insieme delle formule. è definita come segue:

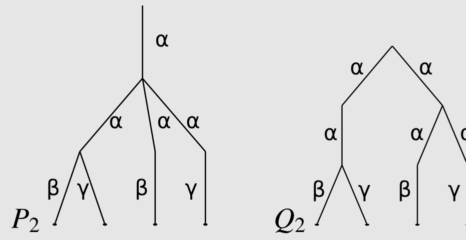
$$\begin{array}{lll} P \models \neg F & sse & non\ P \models F \\ P \models \wedge_{i \in I} F_i & sse & P \models F_i \forall i \in I \\ P \models \diamond_\mu F & sse & \exists p'.\ p \xrightarrow{\mu} p' \quad p' \models F \end{array}$$

Definiamo inoltre:

- $false = \neg true$
- $\vee_{i \in I} F_i = \neg \wedge_{i \in I} \neg F_i$
- $\Box_\mu F = \neg \diamond_\mu \neg F$

in particolare l'ultima implica che per ogni possibile mossa μ (può non essercene nessuna), si va in uno stato in cui vale F.

Esempio 8.29 (Agenti non bisimilari tramite formule logiche)
Dimostrare che i seguenti agente non sono bisimilari fornendo una formula della logica di Hennessy-Milner che distingua P da Q.



$$F = \diamond_\alpha \Box_\alpha (\diamond_\beta true \wedge \diamond_\gamma true)$$

Informalmente: "esiste una transizione α, dove per tutti gli α, abbiamo sia β che γ?"
La formula vale per Q_2 ma non per P_2:

$$Q_2 \models F \qquad P_2 \not\models F$$

Per dimostrare che vale una formula, si usa l'induzione strutturale. Partendo dalle foglie, che sono sempre true, si procede verso l'alto, raggiunta la radice abbiamo gli stati che soddisfano la formula.

Definizione 8.30 (Profondità di una formula)

$$D(\neg F) = D(F)$$
$$D(\wedge_{i \in I} F_i) = smp(D(F_i) \mid i \in I)$$
$$D(\diamond_\mu F) = D(F) + 1$$

Quindi l'insieme delle formule lunghe k viene denotato come

$$\mathscr{L}_k = \{F \mid D(F) = k\}$$

Teorema 8.31
Due agenti sono bisimilari se e solo se soddisfano gli stessi insiemi di formule

$$p \sim_k q \quad sse \quad \forall F \in \mathscr{L}_k (P \models F) \leftrightarrow (Q \models F)$$
$$p \sim q \quad sse \quad \forall F (P \models F) \leftrightarrow (Q \models F)$$

con $\sim_k$ la bisimilarità all'iterazione k.

Dati due agenti bisimilari, questa teorema permette di trovare una proprietà dell'agente più complesso, dimostrandola sull'agente più semplice; non solo, usando la bisimilarità possiamo trovare un agente equivalente *minimo* e tralasciare l'agente complesso, tanto le formule soddisfatte sono le stesse.
Infine la seconda parte del teorema ci assicura che se due agenti non sono bisimilari potremo sempre trovare un controesempio.

8.6. μ-calcolo

Le formule di Hannessy-Milner hanno profondità finita e le visitiamo un passo alla volta, sarebbe utile disporre di operatori di punto fisso per lavorare con formule infinite.
Il μ-calcolo introduce la ricorsione e risulta quindi più potente del CCS conservandone tuttavia le stesse equivalenze.

$$\mu \times .F(X)$$

$$D \times .F(X)$$

QUESTO ESEMPIO NON HA ALCUN SENSO

Esempio 8.32

$$Q \models \Box_\alpha(\Diamond_\beta) true \wedge \Diamond_\gamma true$$

Se uno nega la formula

$$Q \neg \models \Diamond(\Box_\beta false \Box_\gamma false)$$

8.7. Weak Observational equivalence

Il sistema CCS, che permette di osservare le azioni invisibili, non è un modello sufficientemente astratto: in generale vogliamo poter osservare solo le azioni di interesse.
Per nascondere le azioni invisibili τ estendiamo il sistema di transizioni:

Weak Observational Equivalence

$$p \overset{\epsilon}{\Longrightarrow} q \quad sse \quad p \overset{\tau}{\rightarrow} \ldots \overset{\tau}{\rightarrow} q \quad \text{o} \quad p = q$$
$$p \overset{\lambda}{\Longrightarrow} q \quad sse \quad p \overset{\epsilon}{\Rightarrow} p' \overset{\lambda}{\rightarrow} q' \overset{\epsilon}{\Longrightarrow} q$$

Le transizioni sono etichettate solo da stringhe di azioni visibili o dalla stringa vuota.

Utiliziamo la stessa nozione di bisimulazione/bisimilarità:

$$\begin{array}{lllllll} p\Psi(R)q & sse & p \overset{\epsilon}{\Longrightarrow} p' & implica & q \overset{\epsilon}{\Longrightarrow} q' & e & p'Rq' \\ & sse & p \overset{\lambda}{\Longrightarrow} p' & implica & q \overset{\lambda}{\Longrightarrow} q' & e & p'Rq' \\ & & & & & \text{e viceversa} & \end{array}$$

$R \subseteq \Psi(R)$ weak observational equivalencies (i punti fissi) $\approx = \bigcup R$ weak bisimilarity

Purtroppo questa nozione di bisimilarità non è una congruenza, in particolare crea dei problemi per l'operatore "+", come si vede dall'esempio seguente.

Esempio 8.33 (weak bisimilarity non è congruenza)
Mostriamo come due agenti weakly bisimilar se inseriti nello stesso contesto perdono questa proprietà. Da notare che in questo esempio si sfrutti il fatto che Alice e Bob possano intervenire indistintamente su tutti e due gli agenti.

$$\tau\alpha\, \mathbf{nil} \approx \alpha\, \mathbf{nil} \qquad \textit{weak bisimilar}$$

Sono weakly bisimilar perché se Alice fa τ Bob può stare fermo, mentre inseriti nel contesto:

$$\tau\alpha\, \mathbf{nil} + \beta\, \mathbf{nil} \not\approx \alpha\, \mathbf{nil} + \beta\, \mathbf{nil} \qquad \textit{not weak bisimilar}$$

Mosse di Alice e Bob:

$$\begin{array}{llll} \tau\alpha\, \mathbf{nil} + \beta\, \mathbf{nil} & \xrightarrow{Alice:\tau} & \alpha\, \mathbf{nil} & \xrightarrow{Bob:?} \\ \alpha\, \mathbf{nil} + \beta\, \mathbf{nil} & \overset{Bob:\epsilon}{\Longrightarrow} & \alpha\, \mathbf{nil} + \beta\, \mathbf{nil} & \xrightarrow{Alice:\beta} \mathbf{nil} \end{array}$$

L'esempio precedente ci mostra che $\approx$ non è una congruenza, il motivo profondo di questo risultato sta nel fatto che dei τ all'interno di un operatore "+" permettono all'agente di fare delle *scelte invisibili*.
In altri termini, quando analiziamo un sistema in questo modo, come una *black box*, questo oltre che delle mosse interne (invisibili) può anche compiere delle *scelte* interne e questo va al di là dell'astrazione che cercavamo.
Nell'esempio precedente il primo agente fa una scelta "interna" mentre il secondo per compiere la stessa scelta dovrebbe fare una mossa visibile.

8.8. Weak Observational Congruence

Sia $\approx_c$ la più generica congruenza, più piccola della Weak Observational Equivalence. Definiamola rispetto all'uso degli agenti nel contesto dell'operatore "+".

$$p \approx_c q \quad \text{sse} \quad p \approx q \quad \text{e} \quad \forall r.\ p + r \ \approx\ q + r$$

La relazione risultante può essere definita direttamente come

$$\begin{array}{lllll} p \approx_c q & sse & p \xrightarrow{\tau} p' & implica & q \xrightarrow{\tau}\overset{\epsilon}{\Longrightarrow} q' & e \quad p' \approx q' \\ & sse & p \xrightarrow{\lambda} p' & implica & q \overset{\lambda}{\Longrightarrow} q' & e \quad p' \approx q' \\ & & & & & \text{e viceversa} \end{array}$$

In questo modo si elimina la possibilià di stare fermi in risposta da un τ, ma solo per la prima volta.

Questa è la relazione più usata, tuttavia è una congruenza ma non è una bisimulazione rispetto Ψ, cioè

$$\approx_c \not\subseteq \Psi(\approx_c)$$

Infatti basta vedere il seguente controesempio

Esempio 8.34 (Weak Congruence non è bisimulazione)

$$\alpha\tau\beta \ \approx_{wc}\ \alpha\beta$$

applichiamo Ψ, facendo la mossa α abbiamo

$$\tau\beta \ \not\approx_{wc}\ \beta$$

che non sono più nella relazione, infatti il primo può fare τ mentre il socondo no.

I termini sono nella Weak Observational Congruence, ma non sono bisimili, quindi abbiamo la congruenza che ci serviva ma abbiamo perso la bisimulazione.

Inoltre l'operatore $\approx_c$ può essere completamente assiomatizzato dalla *legge τ di Milner*.

$$\begin{array}{l} p + \tau p = \tau p \\ \mu(p + \tau q) = \mu(p + \tau q) + \mu q \\ \mu\tau p = \mu p \end{array}$$

Digressione Se invece ad ogni τ interna corrisponde almeno un τ

$$p\Psi'(\approx)q \text{ iff } \begin{cases} p \xrightarrow{\tau} p' & \text{implies} \quad q \overset{\epsilon}{\Longrightarrow}\xrightarrow{\tau} q' & \text{and} \quad p' \approx q' \\ p \xrightarrow{\lambda} p' & \text{implies} \quad q \overset{\lambda}{\Longrightarrow} q' & \text{and} \quad p' \approx q' \end{cases}$$

In questo caso i due agenti dell'esempio non risulterebbero weakly bisimilar dal principio.

Un altra soluzione potrebbe sembrare quella di introdurre le scelte che abbiamo chiamato "interne" direttamente nel sistema di transizioni, con una regola del tipo

$$p + q \xrightarrow{\tau} p$$

ma questo andrebbe in conflitto con le sincronizzazioni, come mostra l'esempio seguente.

Esempio 8.35

$$(\alpha\,\mathbf{nil} + \beta\,\mathbf{nil}) \mid (\bar{\alpha}\,\mathbf{nil} + \bar{\beta}\,\mathbf{nil})\backslash\alpha\backslash\beta$$

utilizzando il vecchio "+", essendo i canali ristretti, gli agenti possono solo muoversi insieme, mentre con il nuovo il primo potrebbe comunque fare α, il secondo β e finire in deadlock*, come se le restrizioni non ci fossero.*

8.9. Dynamic congruence

La *dynamic congruence* è la più generale (cioè la migliore) relazione che è sia una congruenza che una Ψ-bisimulazione.
Può essere definita direttamente come la più generale relazione $\approx_D$ che soddisfi

$$p \approx_D q \quad \text{implica} \quad \forall C.\ C[p]\ \Psi(\approx_D)\ C[q]$$

e cioè è necessario chiudere con i *contesti* $C[_]$ ad ogni passo.
Oppure semplicemente:

$$\begin{array}{llllll} p\ \Psi''(R)\ q & sse & p \xrightarrow{\tau} p' & \text{implies} & q \xrightarrow{\tau}\overset{\epsilon}{\Longrightarrow} q' & \text{and} \quad p' \approx_D q' \\ & sse & p \xrightarrow{\lambda} p' & \text{implies} & q \xrightarrow{\lambda}\overset{\epsilon}{\Longrightarrow} q' & \text{and} \quad p' \approx_D q' \\ & & & & & \text{e viceversa} \end{array}$$

$$\approx_D = \bigcup_{R \subseteq \Psi''(R)} R \quad \text{dynamic bisimilarity}$$

Rivediamo il controesempio precedente con questa relazione.

Esempio 8.36 (Esempio 8.34 con dynamic)
I due agenti del precedente controesempio non sono nella dynamic congruence dall'inizio e quindi l'esempio risulta corretto.

$$\alpha\tau\beta \not\approx_d \alpha\beta$$

$$\tau\beta \not\approx_d \beta$$

$\approx_d$ può essere assiomatizzato omettendo la terza legge τ di Milner:

$$p + \tau p = \tau p$$
$$\mu(p + \tau q) = \mu(p + \tau q) + \mu q$$

8.10. Caratterizzazione assiomatica della bisimulazione

I calcoli di processo vengono detti anche algebre di processo perché come tutte le algebre sono insiemi con operazioni, ma soprattutto perché la relazione $\simeq_{strong}$ può essere *completamente assiomizzata*.
In prima battuta questo sembrerebbe assurdo perché l'assiomatizzazione è ricorsivamente numerabile, mentre un linguaggio Turing–equivalente non può esserlo, ed infatti l'assiomatizzazione si limita al caso senza ricorsione.

8.10.1. Teorema di Espansione

Dividiamo il processo in due passi

- elimiamo ogni operatore eccetto "+", il prefisso e *nil* tramite gli assiomi di riduzione
- verifichiamo la bisimilarità tramite gli assiomi di bisimilarità strong

8.10.2. Assiomi di riduzione

$$E_1|E_1 = E_1 \llcorner E_2 + E_2 \llcorner E_1 + E_1 \| E_2$$

con $\llcorner E$ a significare che E viene mantenuto fermo. Rompere un parallelo con un'assioma in tre parti.

$$\mu.E_1 \llcorner E_2 = \mu.(E_1|E_2) \begin{cases} \mu_1 E_1 \| \mu_2 E_2 = \mathbf{nil} \quad \text{se} \quad \mu_1 \neq \bar{\mu_2} \\ \lambda E_1 \| \bar{\lambda} E_2 = \tau.(E_1|E_2) \end{cases}$$

Gli operatori $\llcorner$ e $\|$ distribuiscono rispetto al +.

$$(E_1 + E_2)\llcorner E = E_1 \llcorner E + E_2 \llcorner E$$

$$(E_1 + E_1')\|(E_2 + E_2') = E_1 \| E_2' + E_1 \| E_2' + E_1' \| E_2 + E_1' \| E_2'$$

Restrizione:

$$(\mu.E)\backslash\alpha = \begin{matrix} \mu.(E\backslash\alpha) & \text{se} & \mu \neq \alpha, \bar{\alpha} \\ \mathbf{nil} & \text{se} & \mu = \alpha, \bar{\alpha} \end{matrix}$$

$$(E_1 + E_2)\backslash\alpha = E_1\backslash\alpha + E_2\backslash\alpha$$

$$(\mu.E)[\Phi] = \Phi(\mu).E[\Phi]$$

$$(E_1 + E_2)[\Phi] = E_1[\Phi] + E_2[\Phi]$$

In questo modo gli operatori "+" e "." vanno verso l'alto mentre gli altri verso il basso.

8.10.3. Assiomi bisimilarità strong

$$\begin{aligned} E + \mathbf{nil} &= E \\ E_1 + E_2 &= E_2 + E_1 \\ E_1 + (E_2 + E_3) &= (E_1 + E_2) + E_3 \\ E + E &= E \end{aligned}$$

vengono anche detti ACI+nil (Associativity Commutativity Identity).
L'ultimo assioma può sembrare ovvio ma nei sistemi con probabilità ad esempio $E + E \neq E$

TODO-para negli appunti del professore ci sono due esempi di conversione, non credo siano utili.

9. Il Π-calcolo

Un grosso limite del linguaggio appena visto è il fatto che il sistema che ne risulta é decisamente statico. Avremmo bisogno di un sistema che sia

- mobile, cioè nel quale possiamo spostare fisicamente un agente senza dover modificare pesantemente il codice
- dinamico, cioè nel quale possiamo variare durante l'esecuzione la conformazione dei canali
- open-ended, cioè nel quale possiamo attaccarci ad altri sistemi mediante i canali

Il Π-calcolo permette di ottenere questi obiettivi: ci fornisce un ambiente dalla struttura dinamica, nel quale agenti e canali nascono e muoiono di continuo e che permette la mobilità.

Come nel caso di CCS, è buona idea cominciare con un esempio.

Esempio 9.1 (Telefoni mobili)
Consideriamo una struttura come quella visualizzata in Figura 9.1: abbiamo un'automobile con un telefono a bordo che può essere associata a diverse basi che ne gestiranno le telefonate. Le basi sono tutte collegate ad un centro che gestisce le associazioni fra le basi e le automobili.

In alcuni casi il centro può decidere di modificare la morfologia della rete, spostando il controllo dell'automobile da un centro all'altro.

Vediamo le specifiche delle varie parti del sistema.

$$CAR(talk_1, switch_1) \stackrel{\text{def}}{=} \overline{talk}.CAR(talk_1, switch_1) + switch(talk', switch').CAR(talk', switch')$$

Questa è la descrizione della macchina: può dialogare su un canale talk (qua per semplicità assumiamo che possa solamente mandare messaggi) oppure ricevere dal canale switch una coppia di canali nuovi. Notare che il comando porta ad un nuovo stato dove i canali sono stati modificati*: lo stesso canale switch adesso ha un nuovo valore.*

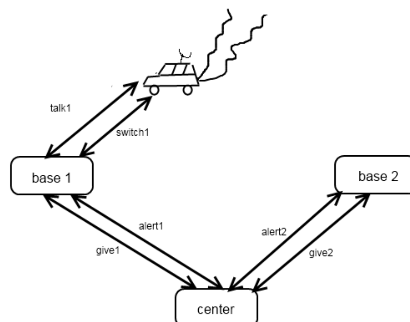


Figura 9.1.: Esempio di struttura nel Π-calcolo

$$BASE_i \stackrel{def}{=} BASE(talk_i, switch_i, give_i, alert_i) \stackrel{def}{=}$$
$$talk_i.BASE_i + give_i(talk', switch').\overline{switch}(talk', switch').IDLEBASE_i$$

Una base attiva ha quattro canali, due dei quali verso la macchina e due verso il centro. Può alternativamente ricevere una comunicazione dalla macchina sul canale talk o ricevere una richiesta di cambio sul canale $give_i$ *da ritrasmettere alla macchina sul canale switch. In questo secondo caso, lo stato cambia e la BASE diventa una IDLEBASE.*

$$IDLEBASE_i \stackrel{def}{=} IDLEBASE(talk_i, switch_i, give_i, alert_i) \stackrel{def}{=} alert_i.BASE_i$$

La IDLEBASE si limita a rimanere in ascolto sul canale alert per sapere quando risvegliarsi. Vediamo il centro operativo: può essere alternativamente in due stati.

$$CENTRE_1 \stackrel{def}{=} CENTRE_1(give_1, alert_1, give_2, alert_2) = \overline{give_1}(talk_2, switch_2).\overline{alert_2}.CENTRE_2$$
$$CENTRE_2 \stackrel{def}{=} CENTRE_2(give_1, alert_1, give_2, alert_2) = \overline{give_2}(talk_1, switch_1).\overline{alert_1}.CENTRE_1$$

I due stati del centro sono perfettamente simmetrici: possono a discrezione dare e togliere il controllo della macchina alle due basi effettuando le giuste comunicazioni. Infine la composizione del sistema è data da

$$SYSTEM = (give_i)(alert_i)(talk_i)(switch_i)i = 1, 2(CAR(talk_1, switch_1)|BASE_1|IDLEBASE_2|CENTRE_1)$$

dove $(give_i)(alert_i)(talk_i)(switch_i)i = 1, 2$ *è la sintassi che verrà usata per restringere i canali.*

9.1. Sintassi di Π-calcolo

9.1.1. Sintassi e potenzialità

Vediamo la sintassi del Π-calcolo.

Assumiamo di avere un insieme $\mathcal{N}$ infinito di nomi, ed utilizziamo i caratteri *x, y, z, w, v, u* e simili come una sorta di meta-variabili su tali nomi.

Presentiamo adesso la sintassi del linguaggio.

$$\begin{array}{rcl} P & ::= & \mathbf{nil} \\ & | & \alpha.p \\ & | & [x = y]p \\ & | & p + p \\ & | & p|p \\ & | & (y)p \\ & | & !p \end{array}$$

$$\begin{array}{rcll} \alpha & ::= & \tau & \text{internal action} \\ & | & x(y) & \text{input} \\ & | & \overline{x}\, y & \text{output} \\ & | & \overline{x}\,(y) & \text{bound output} \end{array}$$

9.1.2. Free names e bound names

Dobbiamo capire bene qual'è il significato di free name e bound name. Vediamo una definizione che ci aiuterà a capire questi due concetti.

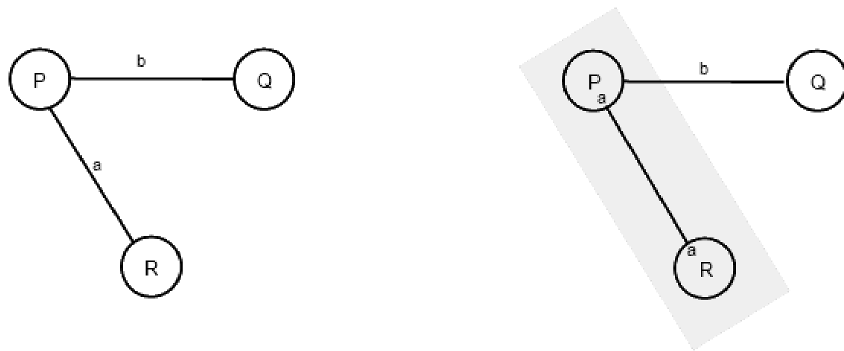


Figura 9.2.: Canali free e bound

Definizione 9.2 (Binding occourrence nel Π-calcolo)
In un agente della forma x(y).p o (y)p, l'occorrenza di y all'interno delle parentesi è detta binding occourrence. *Lo* scope *di tale occorrenza é p.*

Siamo pronti a definire i concetti che vogliamo.

Definizione 9.3 (Free name)
Un nome x è detto free *se non é nello scope di una binding occourrence per x.*

Definizione 9.4 (Bound name)
Un nome é detto bound *se è nello scope di una binding occurrence per x.*

Esistono due maniere dunque per rendere un nome bound: possiamo avere un'operazione di input o una restrizione. Tuttavia, il significato pratico di queste due maniere per bindare il nome è differente.

Nel caso dell'input si tratta semplicemente di dare un nome ad un canale che non abbiamo ancora identificato: è una specie di puntatore a canale. Non ci capiterà mai di maneggiare direttamente un canale il cui nome é bound a causa di un'operazione di input, in quanto l'operazione viene digerita prima di eseguire il corpo e dunque il nome viene istanziato immediatamente.

Il binding per restrizione è completamente diverso: l'utilizzo di questa tecnica é più simile a quello di una variabile di canale che ha una visibilità limitata. Cerchiamo di capire meglio.

Prendiamo l'immagine in figura 9.2, che rappresenta il programma in Π-calcolo con un grafo nel quale i nodi sono processi e gli archi canali di comunicazione. In questa immagine abbiamo a sinistra tre processi paralleli P Q ed R che comunicano attraverso due canali di comunicazione non bound a e b: non esiste nessun operatore di restrizione, e questo significa che tali canali sono free. Se il processo Q utilizzasse al suo interno un canale a, questo sarebbe lo stesso canale che connette i processi P e R: il canale è pubblico e tutti possono scriverci o leggerci.

Tecnicamente, scrivendo i tre processi come

$$\begin{aligned} P &= a(5).P' \\ Q &= \overline{a}\,5.Q' \end{aligned} \qquad\qquad P \quad | \quad Q \quad | \quad R$$

P e Q possono sincronizzarsi: il nome a non è ristretto, dunque può essere utilizzato nelle comunicazioni. Prendiamo invece la situazione di destra: il nome a é bound, e rappresentiamo graficamente questo fatto con

un'area grigia che rappresenta lo scope della restrizione (cioé i processi per i quali il nome é bound) e con il fatto che il nome del canale é scritto all'interno del cerchietto del processo. Scriviamo questa situazione come

$$P = a(5).P' \qquad (a)\,(P \quad | \quad R) \quad | \quad Q$$
$$Q = \overline{a}\,5.Q'$$

Come possiamo vedere, l'operatore di restrizione lega i processi P e R ma non Q. Questo significa che i due processi P e Q non possono sincronizzarsi normalmente: l'operatore di restrizione impedisce che all'esterno si veda la mossa di P. Servirà un meccanismo più complesso che permetta di allargare lo scope del canale a Q. Ovviamente, se esistesse un processo S al di fuori della restrizione che invia dati su un canale x non bound questo potrebbe tranquillamente sincronizzarsi su Q.

9.2. Semantica operazionale del Π-calcolo

Formalizziamo quanto visto nello scorso paragrafo, vedendo le regole di transizione per CCS. Come per quest'ultimo, le formule ben formate sono triple, in cui ci sono due processi p,q ed un'azione α tali che $p \xrightarrow{\alpha} q$.

Dividiamo le regole in paragrafi, in modo da trattarle in maniera più ordinata.

9.2.1. Messaggi

Vediamo innanzitutto le regole per le transizioni

$$\frac{}{\tau p \xrightarrow{\tau} p}$$

$$\frac{}{\overline{x}\,y.p \xrightarrow{\overline{x}\,y} p}$$

Queste due regole sono sostanzialmente invariate rispetto a CCS: semplicemente in caso di una transizione τ (che continua ad indicare un passo di computazione interno) o di output ci si limita a consumarla e passare nello stato consecutivo. Il significato di queste transizioni è rispettivamente quello di eseguire una transizione silenziosa, ovvero senza un'interazione con l'ambiente, o di output, cioé inviando su un canale un certo dato.

Cambia invece qualcosa per quanto riguarda l'input: oltre ai dati, con il π-calcolo è possibile inviare nomi di canali.

$$\frac{}{x(y).p \xrightarrow{x(w)} p[^w/_y]}\; w \notin fn((y)p)$$

Innanzitutto abbiamo il fatto che viene trasmessa un'istanziazione di y, che chiameremo w. Il nome w, e questo è il cambiamento importante, *non deve essere fra i nomi liberi in (y)p*.

Ad esempio, una situazione del genere non va affatto bene.

Esempio 9.5 (Violazione della regola dei nomi in input)
Mettiamo di avere un agente

$$x(y).\overline{z}\,w.nil$$

Effettuiamo la transizione $x(y).\overline{z}\,w.nil \xrightarrow{x(z)} (\overline{z}\,w.nil)[^z/_y]$*: questa non può essere effettuata in quanto z è un nome libero del termine, e questo potrebbe portare a legare una variabile che non dovrebbe essere legata nel caso in cui il z ricevuto fosse all'interno di un operatore di restrizione. Vedremo cosa questo significhi nel paragrafo 9.2.5.*

Notare che questa situazione può essere facilmente risolta con l'α-conversione: in questo caso, basterebbe cambiare il nome del canale z nel processo di destra per non incappare in conflitti. Tuttavia non ci occuperemo di questo, e richiederemo sempre che non ci siano conflitti.

9.2.2. Nondeterminismo

$$\frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

Il comportamento del nondeterminismo è identico a quanto visto per CCS: si sceglie una delle due strade e si *abbandona completamente* l'altra.

9.2.3. Matching

Passiamo al matching.

$$\frac{p \xrightarrow{\alpha} p'}{[x = x]p \xrightarrow{\alpha} p'}$$

Quando troviamo un match, l'*unica* possibilità che abbiamo di andare avanti è il caso in cui i due termini ai lati dell'uguaglianza sono identici. In altro caso, non troviamo prova.

Questo comando è cosiddetto zucchero sintattico: in pratica sarebbe evitabile, ma quando ci si trova ad utilizzarlo é decisamente più comodo. In particolare, é comodo in casi come quello dell'esempio successivo.

Esempio 9.6 (Utilizzo del comando di matching)
Mettiamo di voler scrivere un programma nel quale un processo può svolgere più attività. Un altro processo vuole mandargli un messaggio sul canale c nel quale invoca una di queste attività e gli passa conseguentemente i parametri, ancora sul canale c.

Mettiamo che il processo possa

- *prendere due argomenti x ed y ed eseguire il processo P*
- *prendere un argomento z ed eseguire il processo Q*
- *eseguire il processo R senza argomenti*

Possiamo modellare questa situazione facendo accettare al nostro processo un parametro w che indichi quale processo eseguire, dunque utilizzare nondeterminismo ed operatore di matching per selezionare quale operazione eseguire.

$$c(w).\,([w = p]c(x).c(y).P \quad | \quad [w = q]c(z).Q \quad | \quad [w = r]R)$$

Il programma riceverà w sul canale c ed in base al suo valore (p, q o r) eseguirà l'una o l'altra azione.

9.2.4. Parallelismo

Passiamo al discorso sul parallelismo. Abbiamo innanzitutto due regole quasi identiche a quelle di CCS per far andare avanti una delle due parti del parallelismo.

$$\frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q} \; bn(\alpha) \cap fn(q) = \emptyset \qquad \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'} \; bn(\alpha) \cap fn(p) = \emptyset$$

La differenza sta ancora, come nel caso della ricezione di dati da un canale, nel fatto che possiamo far confusione con i nomi.

$$\frac{p \xrightarrow{\bar{x}\,z} p' \quad q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} p'|(q'[^{z}/_{y}])}$$

Introduciamo infine la regola per la sincronizzazione. L'invio di un dato y sul canale x si lega alla ricezione del dato sullo stesso canale effettuata da un altro processo parallelo e si risolve con la sostituzione del nome della variabile in ingresso con il valore passato dal primo processo.

Vediamo un esempio di come la restrizione sui nomi nella regola sull'avanzamento di una delle due metà del parallelismo sia necessaria.

Esempio 9.7 (Violazione della regola sui nomi nel parallelo)
Mettiamo di avere la regola sull'avanzamento unilaterale vista in CCS.

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

Immaginiamo che $\alpha = x(z)$: con questa regola z verrà scritta in tutto il processo P'. Il problema arriva quando z è contenuta anche nel processo Q: una successiva sostituzione di tale variabile potrebbe creare scompigli non indifferenti!

Per vederlo inseriamo P e Q in un contesto più grande

$$(x(z).P \mid Q) \mid \bar{x}\,y.R$$

Utilizzando le regole di sincronizzazione, input e la regola sull'avanzamento unilaterale vista in CCS (senza dunque vincoli sui nomi) potremmo derivare

$$(x(z).P \mid Q) \mid \bar{x}\,y.R \xrightarrow{\tau} (P \mid Q)[^{y}/_{z}]|R \quad \leftarrow$$
$$(x(z).P \mid Q) \xrightarrow{x(z)} (P \mid Q)[^{y}/_{z}] \qquad \bar{x}\,y.R \xrightarrow{\bar{x}\,y} R \quad \leftarrow$$
$$\Box$$

Come possiamo vedere, la sostituzione di z interessa anche il processo Q, in cui z era una variabile libera qualunque.

Anche qua potremmo risolvere il tutto mediante l'α-conversione, ma non tratteremo questo caso.

9.2.5. Restrizioni, apertura e chiusura

Arriviamo dunque al momento più difficoltoso del Π-calcolo, in quanto momento che piú si discosta dalla visione di mondo di CCS. Vediamo innanzitutto la regola che gestisce la restrizione base.

$$\frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'} \; y \notin n(\alpha)$$

La regola della restrizione vieta l'utilizzo di y in *qualunque modo* all'interno di qualunque azione α, sia come free name che come bound name. Questo comportamento è ancora simile a quello di CCS. Il problema arriva quando dobbiamo allargare il club dei processi che utilizzano una certa variabile privata.

Innanzitutto viene naturale chiedersi perché comunicare un nome bound e perché questo non avvenisse in CCS. La risposta é semplice: in CCS non potevamo comunicare canali, dunque non c'era necessità di gestire la situazione nella quale allargavamo lo scope di un canale privato.

Andiamo a descrivere come questa operazione viene effettuata. Serviranno due regole separate per gestire prima l'apertura del vincolo di restrizione e dunque la chiusura dello stesso con il nuovo processo all'interno.

$$\frac{p \xrightarrow{\overline{x}\, y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'[^w/_y]} \; y \neq x \quad w \notin fn((y)p)$$

La prima regola che vediamo è quella di apertura, che apre il vincolo di restrizione: parliamo dell'operazione di *estrusione*. Si tratta in pratica della trasformazione dell'operazione di output libero in un'operazione di output *legato*: trattasi di un particolare tipo di operazione di output, non direttamente esprimibile nel linguaggio, che indica che il nome che sto trasmettendo sul canale é un nome legato da una restrizione.

Questo operatore elimina l'operatore di restrizione dal processo: questo perché stiamo letteralmente *aprendo* la restrizione per far entrare un altro processo.

$$\frac{p \xrightarrow{\overline{x}\,(w)} p', \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')}$$

L'operazione di chiusura è la complementare dell'operazione di apertura: richiede che i due processi in parallelo eseguano due operazioni complementari, una di lettura e l'altra di scrittura *bound* (e l'unica speranza di generare questa scrittura é con la regola di estrusione), e genera un processo nuovo nel quale la variabile é legata da un operatore di restrizione.

Vediamo un esempio chiarificatore.

Esempio 9.8 (Estrusione di variabile)
Prendiamo il programma

$$(x)(\overline{y}\, x.P' \mid R) \mid y(z).Q'$$

Il processo più a sinistra vuole comunicare al processo a destra la variabile x, che è ristretta. Vogliamo arrivare in una situazione

$$(x)(P'|R|Q'[^x/_z])$$

nella quale lo scope di x sia stato allargato. Vediamo innanzitutto come mai la sincronizzazione normale fallisce e non produce nessuno stato in uscita.

$$(x)(\overline{y}\, x.P' \mid R) \mid y(z).Q' \xrightarrow{\tau} q_1 \xleftarrow{q_1 = q_2 | q_3[^x/_z]} (x)(\overline{y}\, x.P' \mid R) \xrightarrow{\overline{y}\, x} q_2 \quad y(z).Q' \xrightarrow{y(z)} Q'$$

Guardiamo la clausola $(x)(\overline{y}\, x.P' \mid R) \xrightarrow{\overline{y}\, x} q_2$*: richiederebbe di inviare il nome x a partire da un processo per il quale x è ristretta, cosa ovviamente impossibile. L'unica maniera nella quale possiamo procedere é dunque la regola di chiusura, che infatti prende due processi paralleli e li racchiude in una restrizione.*

$$\begin{array}{r} (x)(\overline{y}\, x.P'|R)|y(z).Q' \xrightarrow{\tau} (x)(P'|R|Q'[^x/_z]) \longleftarrow \\ (x)(\overline{y}\, x.P'|R) \xrightarrow{\overline{y}(x)} (P'|R) \quad y(z).Q' \xrightarrow{y(z)} Q' \longleftarrow \\ \overline{y}\, x.P'|R \xrightarrow{\overline{y}\, x} P'|R \longleftarrow \\ \overline{y}\, x.P' \xrightarrow{\overline{y}\, x} P' \longleftarrow \\ \square \end{array}$$

Al primo passo abbiamo applicato la regola di chiusura, trasformando la sincronizzazione in una coppia di operazioni gemelle di invio e ricezione bound. L'unica maniera per derivare un invio bound è con

la regola di apertura, che é stata dunque utilizzata nella terza riga (l'operazione di ricezione é stata banalmente digerita dopo il controllo sui nomi). A questo punto si procede con la regola del parallelo per trovare l'operazione di output e consumarla.

9.2.6. Replicazione

$$\frac{p|!p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

Per ultimo abbiamo l'operatore di replicazione, che ci fornisce un numero illimitato di copie di p.

Vediamo un paio di esempi.

Esempio 9.9 (Derivazione nel Π-calcolo (1))
Prendiamo l'agente del Π-calcolo

$$(((y)\overline{x}\, y.p) \mid q) \mid x(z).r$$

Abbiamo una situazione nella quale il processo $(y)\overline{x}\, y.p$ vuole sincronizzarsi col processo $x(z).r$ stabilendo un canale privato senza che q riesca a sincronizzarsi a sua volta.

Vediamo come si evolve la situazione.

$$(((y)\overline{x}\, y.p) \mid q) \mid x(z).r \xrightarrow{\alpha} q_1 \quad \xleftarrow{q_1=(w)q_2|q_3 \quad \alpha=\tau}$$
$$((y)\overline{x}\, y.p) \mid q \xrightarrow{\overline{x}(w)} q_2 \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \xleftarrow{q_2=q_4|q}$$
$$(y)\overline{x}\, y.p \xrightarrow{\overline{x}(w)} q_4 \qquad w \notin fn(q) \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \xleftarrow{q_4=q_5[^w/_y]}$$
$$\overline{x}\, y.p \xrightarrow{\overline{x}\, w} q_5 \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \xleftarrow{q_3=r[^w/_z] \quad q_5=p} \square$$

Ricostruiamo dunque all'indietro lo stato q_1 nel quale transisco.

$$q_5 = p$$
$$q_4 = q_5\left[^w/_y\right] = p\left[^w/_y\right]$$
$$q_3 = r\left[^w/_z\right]$$
$$q_2 = q_4 \mid q = p\left[^w/_y\right] \mid q$$
$$q_1 = (w)q_2 \mid q_3 = (w)\left(p\left[^w/_y\right] \mid q\right) \mid (r\left[^w/_z\right])$$

In definitiva, possiamo affermare che

$$(((y)\overline{x}\, y.p) \mid q) \mid x(z).r \xrightarrow{\tau} (w)\left(p\left[^w/_y\right] \mid q\right) \mid (r\left[^w/_z\right])$$

Esempio 9.10 (Derivazione nel Π-calcolo (2))
 # DA FARE!!!!

9.3. Semantica astratta di Π-calcolo

9.3.1. Nozioni di equivalenza

Diamo per prima cosa alcune nozioni di equivalenza fra processi. Valgono le seguenti proprietà:

$$\begin{array}{rcc} p + \mathbf{nil} \equiv p & p+q \equiv q+p & (p+q)+r \equiv p+(q+r) \\ p \mid \mathbf{nil} \equiv p & p \mid q \equiv q \mid p & (p \mid q) \mid r \equiv p \mid (q \mid r) \\ (x)\,\mathbf{nil} \equiv \mathbf{nil} & (y)(x)p \equiv (x)(y)p & \\ [x=x]\,p \equiv p & [x=y]\,\mathbf{nil} \equiv \mathbf{nil} & \\ p \mid !p \equiv !p & & \\ (x)(p \mid q) \equiv p \mid (x)q \text{ se } x \notin fn(p) & & \end{array}$$

L'ultima regola è particolarmente importante.
NOTA: vale anche la regola $(x)(p+q) \equiv p + (x)q \quad$ se$x \notin fn(p)$???
è nell'introduzione al pi-calcolo di Parrow pg. 10.

9.3.2. Semantica astratta

Vediamo adesso la semantica astratta del Π-calcolo. Come per CCS ci baseremo sulla nozione di bisimilarità, ed anche qua avremo la possibilità di definire una bisimilarità *strong* ed una *weak*. Ci concentreremo solamente su bisimulazioni strong, in quanto come vedremo avremo problemi di congruenza già con queste. Vedremo prima le versioni ground, che ci daranno problemi di congruenza, e quindi passeremo ad una loro evoluzione che risolverà questi problemi.

9.3.2.1. Bisimularità ground

Siamo pronti a vedere la bisimularità di tipo *ground* per il Π-calcolo.

Per quanto riguarda le operazioni τ e di output, la bisimularità è identica al caso visto in CCS

$$pSq \Leftrightarrow p \xrightarrow{\alpha} p' \quad \alpha \neq x(y) \quad \wedge \quad bn(\alpha) \notin fn(q) \qquad \Rightarrow q \xrightarrow{\alpha} q' \quad \wedge \quad p'Sq' \quad \text{e viceversa}$$

Per quanto riguarda le operazioni di input la storia cambia. Dobbiamo assicurare l'uguaglianza per *qualunque* valore passiamo sul canale: per assicurare questa proprietà possiamo seguire due strade.

La prima, che ci darà una semantica cosiddetta *early*, richiede che per ogni possibile nome in ingresso si trovi uno stato q' che soddisfi l'uguaglianza.

$$pSq \Leftrightarrow p \xrightarrow{x(y)} p' \quad y \notin fn(q) \quad \Rightarrow \quad \forall w.\exists q'.q \xrightarrow{x(y)} q' \quad p'[^w/_y]Sq'[^w/_y] \text{ e viceversa}$$

La bisimilarità early si indica con il simbolo $\overset{\circ}{\sim}_E$.

La seconda, che ci darà una semantica cosiddetta *late*, richiede che esista un unico stato q' che verifica l'uguaglianza per qualunque nome in ingresso sul canale.

$$pSq \Leftrightarrow p \xrightarrow{x(y)} p' \quad y \notin fn(q) \quad \Rightarrow \quad \exists q'.\forall w.q \xrightarrow{x(y)} q' \quad p'[^w/_y]Sq'[^w/_y] \text{ e viceversa}$$

La bisimilarità late si indica con il simbolo $\overset{\circ}{\sim}_L$.

Notare che entrambe le bisimilarità hanno la regola per τ e output in comune. In generale la bisimularità early tenderà a distinguere meno rispetto a quella late, in quanto per nomi diversi in ingresso possiamo scegliere stati diversi. Vediamo un esempio in cui questo porta a valutare differentemente un caso.

Esempio 9.11 (Bisimularità early e late)
Prendiamo i due processi

$$p = x(y).\tau.\,\mathbf{nil} + x(y).\,\mathbf{nil}$$
$$q = p + x(y).[y=z].\tau.\,\mathbf{nil}$$

Vediamo perchè i processi p e q risultano equivalenti per la bisimulazione early e non quella late:

- *Nel caso early dobbiamo verificare che per ogni nome in ingresso, esista uno stato q' tale che simuli p' e viceversa.*
  - *se q transisce in $x(y).[y = z].\tau.nil$ dobbiamo trovare uno stato in p tale per cui, nel caso $u = z$, si comporti come $\tau.nil$: in questo caso va benissimo $x(y).\tau.$ **nil**.*
  - *per valori diversi da z dobbiamo andare in stallo, a questo punto sceglieremo $x(y).$ **nil** che fa esattamente quello che chiedevamo*
- *Nel caso della late invece dobbiamo trovare (almeno) uno stato che emuli perfettamente il comporamento di $x(y).[y = z].\tau.$ **nil**, ma questo non è possibile.*

In sostanza i due processi risultano equivalenti per $\overset{\circ}{\sim}_E$ ma non per la $\overset{\circ}{\sim}_L$, infatti se scegliamo come $q' = x(y).[y = z].\tau.nil$ e poi usiamo un $y \neq z$ abbiamo un deadlock che in p non si presenta.

9.3.2.2. Bisimilarità non ground

La bisimilarità ground è troppo sempliciotta, ed infatti non é una congruenza rispetto al prefisso di input. Vediamo un controesempio

Esempio 9.12 (La bisimilarità ground non è una congruenza)
É possibile dimostrare che gli agenti

$$p = \overline{x}\, x.nil \mid x'(y).nil \qquad q = \overline{x}\, x.x'(y).nil + x'(y).\overline{x}\, x.nil$$

sono bisimilari. In particolare, nessuno dei due è capace di eseguire una sincronizzazione e dunque di transire con τ.

Proviamo ad inserirli in un contesto $z(x')(\ldots)$. I due agenti risultanti non sono bisimilari*: il motivo è che all'esterno mettiamo qualcosa che* lega una variabile precedentemente libera*, il che può portare a risultati scorretti.*

$$p' = z(x')(\overline{x}\, x.nil \mid x'(y).nil) \qquad q' = z(x')(\overline{x}\, x.x'(y).nil + x'(y).\overline{x}\, x.nil)$$

se come x' inviamo proprio x, in questo caso, il primo agente può eseguire una mossa $z(x)$ andando nello stato $\overline{x}\, x.nil \mid x(y).nil$, per poi fare una sincronizzazione τ sul canale x, mentre l'altro non lo può fare.

Il problema è quello dell'*aliasing*: inserire un agente valido all'interno di un contesto può legare nomi che precedentemente erano liberi. Per evitare che questo crei problemi con le congruenze, abbiamo bisogno di una bisimilarità più stretta.

Definiamo prima formalmente il concetto di sostituzione in CCS e di applicazione di sostituzione.

Definizione 9.13 (Sostituzione)
Una sostituzione è una funzione $\sigma : \mathcal{N} \to \mathcal{N}$, dove $\mathcal{N}$ é l'insieme dei nomi, che é quasi ovunque l'identità (ovvero che é diversa dall'identità in un numero finito di casi).

Indichiamo l'applicazione ad una variabile con $x_i\sigma$. Se abbiamo una sostituzione tale che $x_i\sigma = y_i$ con $1 \leq i \leq n$ (e $x\sigma = x$ per tutti gli altri nomi) possiamo scrivere $[^{y_1}/_{x_1}, \ldots, ^{y_n}/_{x_n}]$.

Abbiamo già usato la sostituzione in alcune regole in precedenza. Vediamo come la sostituzione si applica

Definizione 9.14 (Applicazione di sostituzione)
Indichiamo con $P\sigma$ l'agente P al quale è stata applicata una sostituzione σ. Questa sostituzione é ottenuta sostituendo simultaneamente *$z\sigma$ ad ogni variabile libera z in P, cambiando i nomi delle variabili legate per evitare problemi.*

Definiamo dunque la bisimilarità non ground.

Definizione 9.15 (Bisimilarità early non ground)
Dati due agenti p e q, ho

$$p \sim_E q \Leftrightarrow \forall \sigma . p\sigma \overset{\circ}{\sim}_E q\sigma$$

Definizione 9.16 (Bisimilarità late non ground)
Dati due agenti p e q, ho

$$p \sim_L q \Leftrightarrow \forall \sigma . p\sigma \overset{\circ}{\sim}_L q\sigma$$

9.4. Estensioni del Π-calcolo

Nel tempo sono nate alcune estensioni che miravano ad estendere, o restringere, il *Pi*- calcolo. è interessante vedere che queste estensioni non aumentano né diminuiscono il potere espressivo.

9.4.1. Π-calcolo asincrono

Il Π-calcolo asincrono è una restrizione del Π-calcolo che richiede che le comunicazioni siano asincrone. Dopo aver effettuato un'operazione di output, un agente può avere solamente nil come azione.

É possibile simulare il Π-calcolo normale nella seguente maniera: data una generica sincronizzazione $\overline{x}\, y.P \mid x(y).Q$ la sostituisco con

$$\underbrace{(z)\overline{x}\, z.nil}_{1} \mid \underbrace{\overline{z}\, y.nil}_{2} \mid \underbrace{z(z).P}_{3} \mid \underbrace{x(z).z(y).(\overline{z}\, z \mid Q)}_{4} \qquad z \notin fn(P, Q)$$

per prima cosa il partner 1 spedisce il nome privato z sul canale x. Questo viene catturato dal partner 4, e questa sincronizzazione sblocca l'uso del canale z: viene quindi eseguita la sincronizzazione con 2, e viene dunque mandato il nome z sul canale z che viene raccolto da 3 che può finalmente cominciare l'esecuzione di P. In parallelo viene eseguito Q, come previsto.

9.4.2. HOPI: High-order Π-calculus

Più sorprendente è il risultato che viene tentando di estendere il Π-calcolo con funzioni di ordine superiore al primo. Possiamo trasmettere adesso interi processi attraverso un canale! Può sembrare un cambiamento epocale, eppure ci rendiamo conto che il potere espressivo é lo stesso del Π-calcolo classico. Infatti una transizione $\overline{x}R.P \mid x(Y).Y$ puó essere simulata con

$$(z)\overline{x}\, z.(z(z).R \mid P) \mid x(z).\overline{z}\, z.nil$$

Al posto del processo viene mandato un puntatore ad un trigger che avvia il processo stesso.

Parte IV.

Calcoli di processo probabilistici

10. Teoria della misura e catene di Markov

Passiamo adesso a vedere l'ultima classe di linguaggi del corso: i linguaggi per lo studio di modelli probabilistici.

Sono linguaggi basati su CCS, opportunamente esteso con costrutti probabilistici. Questo porterà ad un grafo di esecuzione rappresentabile da una catena di Markov.

Prima di addentrarci nella trattazione del linguaggio dobbiamo avere qualche base matematica: rivedremo alcuni aspetti della teoria della misura per poi confrontarci con le catene di Markov e, nel prossimo capitolo, capire come utilizzarle per definire un linguaggio che ci aiuti a far quello di cui abbiamo bisogno.

10.1. La teoria della misura

10.1.1. I Sigma-field

La teoria della misura, che riguarda la probabilità, è il primo passo da fare verso le catene di Markov. Partiamo dal primo tassello di questa teoria: il concetto di *sigma-field*, o sigma-algebra.

Definizione 10.1 (Sigma Field (σ-algebra))
Un sigma field è formato da

- *un insieme* Ω *di oggetti*
- *un insieme* $F \subseteq 2^{\Omega}$ *di sottoinsiemi di* Ω*, tale che*
  - $\varnothing \in F$
  - $\Omega \in F$
  - $I \in F \Rightarrow F \setminus I \in F$ *(F è chiuso rispetto al complemento)*
  - $I_1, I_2 \in F \Rightarrow I_1 \cup I_2 \in F, I_1 \cap I_2 \in F$ *(chiuso rispetto ad unione ed intersezione numerabile)*

La coppia Ω, F è detta *spazio misurabile*. Questa rappresenta l'insieme di oggetti sui quali costruiremo la nostra *funzione di probabilità*, che definiremo a breve.

Esempio 10.2
Se Ω *= a, b, c, d, un possibile insieme F per formare un sigma field potrebbe essere*

$$F \subseteq 2^{\Omega} = \{\varnothing, \{a, b\}, \{c, d\}, \{a, b, c, d\}\}$$

Sopra i sigma field appena descritti definiamo le misure, cioè metodi sistematici per assegnare ad ogni sottoinsieme una specie di punteggio, rappresentato da un numero reale tra 0 e ∞.

Definizione 10.3 (Misura su (Ω, F))
Una misura su (Ω, F) *è una funzione del tipo*

$$\mu : F \to R \cup \{\infty\}$$

tale che

$$\mu(\emptyset) = 0$$
$$\mu(\bigcup_i X_i) = \Sigma_i \mu(X_i) \quad \textit{se } X_i \textit{ disgiunti}$$
$$\mu(\Omega)$$

Vediamo due particolari misure che ci interesseranno particolarmente.

Definizione 10.4 (Sub-probability)
Una sub-probability *è una misura tale che* $\mu(\Omega) \leq 1$

Definizione 10.5 (Probability)
Una sub-probability *è una misura tale che* $\mu(\Omega) = 1$

10.1.2. Costruire un sigma-field

Una maniera se vogliamo classica di costruire un sigma-field è quella di definirlo a partire da una famiglia di sottoinsiemi che chiuderemo rispetto a unione, intersezione e complemento.

Tuttavia, le cose sono leggemente più complesse a seconda della cardinalità di Ω: se questi ha una cardinalità finita o numerabile non ci sono problemi, mentre se ha una cardinalità del continuo l'insieme F potrebbe avere la cardinalità $\aleph_2$ delle funzioni, che sarebbe un insieme troppo grande per definirvi sopra una misura. In caso di insiemi con la cardinalità del continuo dovremo dunque scegliere un sottoinsieme della stessa cardinalitá di sottoinsiemi di lavoro.

Vediamo un esempio nel quale questo concetto trova applicazione: abbiamo un insieme Ω troppo grande e dobbiamo ridurlo prima di definirvi una misura di probabilità.

Esempio 10.6 (Cammini finiti)
Immaginiamo di avere a disposizione una moneta non truccata. Vogliamo rappresentare tutte le sequenze di lanci possibili. Viene naturale definire il nostro insieme Ω *come chiusura rispetto al prodotto cartesiano dell'insieme*

$$\Omega = \{h, t\}^\infty$$

Tuttavia, questo insieme ha la cardinalità del continuo: i suoi sottoinsiemi sono infiniti.

Lo stratagemma che utilizzeremo consiste nel prendere tutti i sottoinsiemi di cammini finiti *di* Ω. *I cammini finiti sono stringhe in un alfabeto di due caratteri, e sono un insieme numerabile che ci va benissimo.*

Per non perdere informazione, consideriamo anche che un cammino finito individui un particolare albero infinito di cammini che hanno tale cammino come prefisso

$$[\alpha] = \{\alpha\beta | \beta \in \Omega\}$$

L'altro metodo per risolvere il problema è quello di utilizzare una topologia. Vediamo cosè.

Definizione 10.7 (Topologia)
Sia X un insieme e sia τ *una famiglia di sottoinsiemi di X. Allora* τ *è una* topologia *su X sse X e* $\tau \subseteq 2^X$ *sono* open sets, *cioè insiemi tali che*

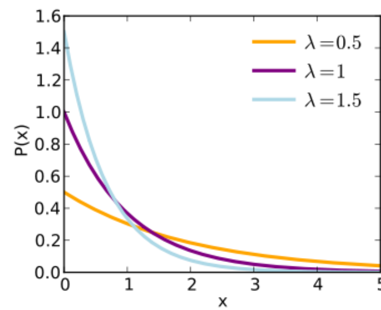


Figura 10.1.: Densità della distribuzione esponenziale

- $X, \emptyset \in \tau$
- *chiuso per unione: cioè l'unione di un numero arbitrario di elementi di τ è ancora un elemento di τ*
- *chiuso per intersezione finita: cioè l'intersezione di un numero arbitrario di elementi di τ è ancora un elemento di τ*

Intuitivamente un insieme U è *aperto* se ogni suo punto può essere mosso in una qualsiasi direzione ed essere ancora in U. è possibile chiudere una topologia ottenendo un sigma field.

Se τ è una topologia su X allora la coppia (X, τ) è detto uno *spazio topologico*.

Definizione 10.8 (Topologia euclidea)
Intervalli aperti

$$]a, b[\quad = \quad \{x \in \mathbb{R} \mid a < x < b\}$$

Se chiudiamo la topologia euclidea otteniamo il Sigma field di Borel.

Ci interesseranno i casi discreti. Se abbiamo il caso continuo, dobbiamo considerare i casi aperti e i cammini lungo un sistema di transizione.

10.1.3. Distribuzione probabilistca esponenziale

La distribuzione probabilistica esponenziale ci servirà per descrivere il comportamento delle nostre catene di Markov. Rivediamone i concetti principali.

É una distribuzione definita sui valori positivi, che ha funzione di ripartizione $F(x) = 1 - e^{-\lambda x}$ e densità $d(x) = \lambda e^{-\lambda x}$: il grafico è riportato in figura 10.1. è una distribuzione *senza memoria*: vediamo cosa questo significhi.

Esempio 10.9 (Distribuzione senza memoria)
Abbiamo un sistema in uno stato S_1, che può decadere e passare in uno stato S_2. Un valido esempio può essere un atomo radioattivo.

$$S_1 \rightarrow S_2$$

Vogliamo modellare la probabilità di questo evento, considerando che è completamente slegato dal tempo: la probabilità che l'atomo decada in un tempo $t_0 + t$ sapendo che al tempo t_0 non é decaduto é la stessa per qualunque tempo t_0.

Dimostriamo le due proprietà che ci interessano. La prima è quella più importante: la distribuzione esponenziale é senza memoria.

Teorema 10.10 (La distribuzione esponenziale è senza memoria)
Sia d una variabile casuale continua che indica il tempo al quale avviene un certo avvenimento. Se tale variabile segue una legge esponenziale di parametro λ, vale

$$Prob\{d \le t_0 + t \mid delay > t_0\} = Prob\{d \le t\}$$

Dimostrazione. La variabile d segue una distribuzione esponenziale: utilizziamo la sua funzione di ripartizione ottenendo

$$\frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda t}}{\int_{t_0}^{\infty} \lambda e^{-\lambda t}} = \int_0^t \lambda e^{\lambda t}$$

Vale $\int_a^b \lambda e^{-\lambda t} = \left[-e^{-\lambda t}\right]_a^b = \left[e^{-\lambda t}\right]_b^a$: quindi abbiamo a sinistra

$$\frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda t}}{\int_{t_0}^{\infty} \lambda e^{-\lambda t}} = \frac{\left[e^{\lambda t}\right]_{t_0+t}^{t_0}}{\left[e^{-\lambda t}\right]_{\infty}^{t_0}} = \frac{e^{-\lambda t_0} - e^{\lambda t} \cdot e^{-\lambda t_0}}{e^{-\lambda t_0}} = \frac{\cancel{e^{-\lambda t_0}}(1 - e^{\lambda t})}{\cancel{e^{-\lambda t_0}}} = 1 - e^{\lambda t}$$

e a destra

$$\int_0^t \lambda e^{\lambda t} = \left[e^{\lambda t}\right]_t^0 = 1 - e^{\lambda t}$$

La seconda proprietà riguarda la situazione in cui ci troviamo in uno stato dal quale ci sono due o più possibili transizioni in uscita e vogliamo sapere la probabilità di aver seguito una *qualunque* transizione prima di un certo tempo t.

Teorema 10.11 (Transizioni multiple)
Dati due eventi, siano d_1 e d_2 due variabili che contengono l'istante di tempo nel quale questi due eventi accadono. Se tali variabili seguono una legge esponenziale di parametro rispettivamente λ_1 e λ_2, vale

$$Prob\{\min\{d_1, d_2\} \le t\} = 1 - e^{-(\lambda_1+\lambda_2)t}$$

Dimostrazione. Si vede semplicemente che

$$\begin{aligned} &Prob\{delay_1 \le t\} + Prob\{delay_2 \le t\} - Prob\{delay_1 \le t \wedge delay_2 \le t\} \\ &= (1 - e^{-\lambda_1 t}) + (1 - e^{-\lambda_2 t}) - (1 - e^{-\lambda_1 t})(1 - e^{-\lambda_2 t}) = 1 - e^{-\lambda_1 t} \cdot e^{-\lambda_2 t} = 1 - e^{-(\lambda_1+\lambda_2)t} \end{aligned}$$

La terza proprietà serve a confrontare due eventi.

Teorema 10.12 (Confronto di due eventi)
Dati due eventi, siano d_1 e d_2 due variabili che contengono l'istante di tempo nel quale questi due eventi accadono. Se tali variabili seguono una legge esponenziale di parametro rispettivamente λ_1 e λ_2, vale

$$Prob\{delay_1 < delay_2\} \stackrel{?}{=} \frac{\lambda_1}{\lambda_1 + \lambda_2}$$

Dimostrazione. Anche qua si dimostra semplicemente nella seguente maniera

$$\int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \left(\int_{t_1}^\infty \lambda_2 e^{-\lambda_2 t} dt_2 \right) dt_1$$
$$= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \left[e^{-\lambda_2 t_2} \right]_\infty^{t_1} dt_1$$
$$= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \cdot e^{-\lambda_2 t_1)}$$
$$= \int_0^\infty \lambda_1 e^{-(\lambda_1+\lambda_2) t_1} dt_1$$
$$= \left[\frac{\lambda_1}{\lambda_1+\lambda_2} e^{-(\lambda_1+\lambda_2)t} \right]_\infty^0$$
$$= \frac{\lambda_1}{\lambda_1+\lambda_2}$$

10.2. Processi Stocastici Markoviani

10.2.1. Le catene di Markov

Introduciamo adesso i processi stocastici, che rappresenteremo con delle catene di Markov. Cosa sono queste catene? Vediamo di arrivarci.

Immaginiamo di avere una linea del tempo, nella quale prendiamo un insieme di istanti di tempo $t_1, \dots, t_n$. Ad ognuno di questi istanti associamo una variabile casuale $X_{t_i} : S \to \mathbb{R}$: S sarà ovviamente un sigma-field, e completamente indipendente da chi sono i punti t_i.

Il significato della variabile casuale X_{t_i} è quello di esprimere la probabilità di trovarsi in un dato stato nell'istante t_i. Un sistema del genere é detto processo stocastico.

Definizione 10.13 (Catena di Markov)
Una catena di Markov, o processo stocastico Markoviano, è un processo stocastico per il quale vale la Proprietà di Markov

$$Prob\{X_{t_{n+1}} \in A \mid X_{t_n} = P_n, \dots, X_{t_0} = P_0\} = Prob\{X_{t_{n+1}} \in A \mid X_{t_n} = P_n\}$$

In pratica, in un processo Markoviano la probabilità che $X_{t_{n+1}}$ goda di una certa proprietà dipende solo dallo stato X_{t_n} e non da tutti gli altri precedenti.

Le proprietà che possiamo misurare su tali stati sono numerose: ad esempio, possiamo vedere se in quel momento ci troviamo in un particolare insieme di stati o simili.

In generale vogliamo che tutta la storia passata sia riassunta nel solo immediatamente stato precedente: se questo non accade si parla di *processo non Markoviano*. In quel caso, per poter trattare il processo stocastico come se fosse una catena di Markov dobbiamo immagazzinare all'interno dello stato un'informazione sulla sua storia.

Esistono alcune particolari catene di Markov che possono essere trattate in maniera decisamente più semplice perdendo un po' di flessibilità. Vediamole.

Definizione 10.14 (Catena di Markov omogenea)
Una catena di Markov è detta omogenea se vale

$$Prob\{X_{t'}(t' > t) \in A | X_t = P\} = Prob\{X_{t'-t} \in A | X_0 = P\}$$

Questo significa che qualunque t' può essere translato su $t' - t$, e più in generale è possibile traslare l'asse dei tempi a piacimento. Questa proprietà ci permette di essere indipendenti dal tempo e immaginare di partire sempre dall'istante zero ad esempio.

10.2.2. Catene di Markov a tempo discreto e continuo

Nei prossimi paragrafi vedremo due insiemi di catene di Markov: quelle con a tempo *discreto* e quelle a tempo *continuo*. Cerchiamo di capire quale sia la differenza fra queste catene e quale sia meglio usare per modellare una situazione.

Cominciamo con il definire le catene di Markov a tempo discreto

Definizione 10.15 (Discrete-time Markov Chain (DTMC))
Una DTMC è una catena di Markov nella quale l'insieme degli istanti di tempo é numerabile.

Questo ci permette di avere il concetto di istante precedente nella nostra catena di Markov: possiamo dunque calcolare una proprietà in un istante considerando com'era il sistema nello stato precedente.

$$Prob\{X_{n+1} \in A | X_n = P_n\}$$

Vedremo cosa questo significhi nella sezione 10.3.

Se invece l'insieme degli istanti di tempo è continuo abbiamo una Continuous Time Markov Chain.

Definizione 10.16 (Continuous-time Markov Chain (CTMC))
Una DTMC è una catena di Markov nella quale l'insieme degli istanti di tempo é continuo.

Questo ci porterà a dire che la probabilità che ad un certo istante Δt valga una certa proprietà t' sarà proporzionale all'intervallo di tempo Δt. Approfondiremo nella sezione 10.4.

10.3. DTMC - Discrete-time Markov Chain

10.3.1. Relazioni con CCS

Le catene di Markov discrete sono un sistema di transizioni molto simile al CCS: abbiamo un insieme di stati (possibilmente infinito) e delle transizioni. La differenza fondamentale è che le transizioni non sono etichettate da azioni, ma da probabilità.

Rivediamo due espressioni equivalenti della funzione di transizione di CCS.

$$\alpha : S \to \mathscr{P}(L \times S) \qquad \alpha : \mathscr{P}(S \times L \times S)$$

Possiamo vedere α come una funzione che dato lo stato di partenza ritorna le coppie contenenti l'etichetta e lo stato di arrivo, o equivalentemente come un insieme di triple che contengono le stesse informazioni.

$$\alpha : (S \times L) \to \mathscr{P}(S) \qquad \alpha : S \to L \to \mathscr{P}(S)$$

Alternativamente possiamo vederla come una funzione che prende uno stato di partenza ed una etichetta come input e ritorna l'insieme degli stati di arrivo (notare che sono insiemi perché il sistema è non deterministico).

Scopriremo a breve che queste espressioni, che per CCS sono assolutamente equivalenti, portano a qualche cambiamento nelle nostre catene di Markov.

10.3.2. Definizioni

Vediamo come funziona la funzione di transizione nel caso delle catene di Markov discrete.

$$\alpha : S \to (D(S) + 1) \qquad \text{Unione disgiunta con singleton}$$

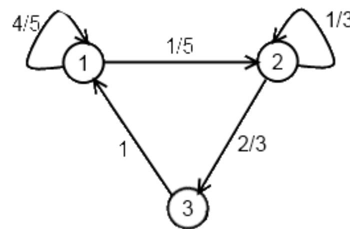


Figura 10.2.: Un esempio di Discrete-time Markov Chain

Abbiamo una funzione che, dato uno stato s, restituisce una *distribuzione di probabilità* che ci dice, se al tempo t mi trovo in s, con quale probabilità mi troverò in un certo stato s' al tempo t+1. Alternativamente, posso segnalare uno stato di *deadlock* dal quale non ho transizioni in uscita. è questo il ruolo del +1 nel codominio della funzione.

Per ogni nodo deve valere che la somma degli archi uscenti faccia uno.

$$\sum_i \alpha(s)(s_i) = 1 \qquad \text{con } \alpha(s) \in D(S)$$

Vediamo un esempio.

Esempio 10.17 (DTMC)
Prendiamo la DTMC in figura 10.2, che rappresentiamo con un grafo diretto e pesato. La stessa situazione può essere espressa in forma matriciale nella seguente maniera.

$$\begin{vmatrix} 0.8 & 0 & 1 \\ 0.2 & 1/3 & 0 \\ 0 & 2/3 & 0 \end{vmatrix}$$

Vogliamo calcolare la probabilità di trovarci in un certo stato al tempo t_1 assumendo di non sapere in quale stato siamo precedentemente. Supponiamo di avere $\frac{1}{3}$ di probabilità di trovarci in un particolare stato: dobbiamo eseguire una moltiplicazione di matrice, come vediamo qua sotto.

$$\begin{vmatrix} 1/3 \\ 1/3 \\ 1/3 \end{vmatrix} \begin{vmatrix} 0.8 & 0 & 1 \\ 0.2 & 1/3 & 0 \\ 0 & 2/3 & 0 \end{vmatrix} = \begin{vmatrix} 5.4/9 \\ 4.5/9 \\ 2/9 \end{vmatrix}$$

In questo esempio, il vettore di partenza pone di partire da uno qualunque degli stati in modo del tutto equiprobabile, mentre il risultato del calcolo mostra le probabilità di trovarsi in uno stato dopo una mossa dell'agente. Se moltiplicassimo il vettore risultante ancora per la matrice, otterremmo le probabilità al passo ancora successivo e così via.

Vediamo un'altra differenza rispetto a CCS. In CCS era perfettamente logico avere due transizioni differenti che collegassero gli stessi stati: ad esempio l'agente $\mu.P + \mu.P$ ha due transizioni sintatticamente diverse fra se stesso e lo stato P.

Nelle catene di Markov questo non ha affatto senso: qua due azioni che collegano gli stessi nodi con probabilità rispettivamente a e b possono essere unite in una transizione unica, come vediamo in figura 10.3.

Vediamo adesso una definizione che ci indica come si calcola la probabilità di seguire un certo cammino all'interno del nostro sistema.

(a) (b)

Figura 10.3.: Due DTMC equivalenti: la seconda è ricavata sommando i coefficienti della prima

Definizione 10.18 (Probabilità di un cammino)
Definiamo la probabilità di un cammino come

$$Prob(s_1 \ \ldots \ s_n) = \Pi_{i=1}^{n-1} \lambda_{s_i \, s_{i+1}}$$

Vediamo un esempio

Esempio 10.19 (Probabilità di un cammino)
Ad esempio, prendendo la matrice dell'esempio 10.17, abbiamo

$$Prob(1\ 2\ 3\ 1) = 0.2 \cdot \frac{2}{3} \cdot 1 = \frac{2}{15}$$
$$Prob(1\ 1\ 3\ 1) = 0.8 \cdot 0 \cdot 1 = 0$$

In caso di cammini infiniti ci possiamo comportare come abbiamo visto nell'esempio 10.6.

Esempio 10.20 (Random walk)
Immaginiamo di avere un asse come lo spazio per il nostro cammino e che il camminatore parta dallo zero per spostarsi a destra o sinistra di un'unità, con uguale probabilità.

$$P(0) = 1$$
$$m \xrightarrow{1/2} m - 1$$
$$m \xrightarrow{1/2} m + 1$$

Il numero di possibili cammini che il camminatore può effettuare è infinito, così come la lunghezza di alcuni di tali cammini. Rappresentiamo i cammini con i loro prefissi finiti, in modo da non incappare in problemi di cardinalità.

Notare che, per la legge dei grandi numeri, immaginando di prendere l'insieme dei cammini che da un certo punto in poi non toccano più lo zero la probabilità di tutto questo insieme è zero.

10.3.3. Limite di una catena: catene ergodiche

Viene da chiedersi: se partiamo da uno stato qualsiasi e continuiamo ad applicare la matrice, è possibile che ad un certo punto si arrivi ad un punto stabile?

Scopriamo che questo accade per le catene di Markov Ergodiche.

Definizione 10.21 (Catene ergodiche)
Una catena di Markov è detta ergodica *quando*

i) *è irriducibile: per ogni coppia di stati deve esistere un cammino che li colleghi.*

ii) *è aperiordica: il MCD della lunghezza dei cicli deve essere 1.*

Abbiamo che i) implica che non ci siano deadlock mentre ii) che non ci siano cammini periodici, ad esempio se togliessimo i *self-loop* dall'esempio 10.17 avremmo un sistema periodico.

Questo limite sarà della forma

$$S \times M = S \qquad \sum_{s_i} = 1$$

Con una catena di Markov ergodica abbiamo una ed una sola soluzione detta *spazio stazionario*.

Esempio 10.22 (Punto di stabilizzazione)
Nell'esempio 10.17 il punto nel quale le probabilità si stabilizzano è $S_1 = 2/3 \quad S_2 = 1/5 \quad S_3 = 2/15$. *Infatti*

$$\begin{vmatrix} 2/3 \\ 1/5 \\ 2/15 \end{vmatrix} \begin{vmatrix} 0.8 & 0 & 1 \\ 0.2 & 1/3 & 0 \\ 0 & 2/3 & 0 \end{vmatrix} = \begin{vmatrix} 2/3 \\ 1/5 \\ 2/15 \end{vmatrix}$$

10.4. CTMC - Continuous-time Markov Chain

Vediamo adesso il funzionamento di una CTMC. Adesso abbiamo un tempo continuo: non possiamo più dunque calcolare la probabilità di essere in uno stato rispetto allo stato precedente in quanto non esiste piú neppure il concetto di istante di tempo precedente.

Le transizioni uscenti saranno dunque etichettate con *rate della distribuzione esponenziale* che ci da la probabilità che in un certo tempo avvenga un cambiamento di stato lungo quella transizione.

10.4.1. Definizioni

Vediamo la funzione di transizione per CTMC: può essere espressa alternativamente in una di queste forme

$$\alpha : S \rightarrow \mathscr{P}(\mathbb{R} \times S)$$

$$\alpha : (S \times S) \rightarrow \mathbb{R}$$

Abbiamo ancora un grafo che è etichettato con i rate della distribuzione esponenziale. Per capire come lo stato varia, dobbiamo utilizzare le relazioni viste nella sottosezione 10.1.3.

Anche nelle CTMC è vero che due transizioni fra gli stessi stati con rate rispettivamente λ_1 e λ_2 sono equivalenti ad una transizione sola con rate $\lambda_1 + \lambda_2$. è inoltre vero che le transizioni che collegano uno stato a se stesso sono trascurabili: nelle DTMC rappresentavano il fatto che a due istanti di tempo successivi potevamo trovarci nello stesso stato, mentre qua perdono anche questo senso diventando di fatto inutili.

Possiamo sempre trasformare una CTMC in una DTMC, pur perdendo informazione su quanto tempo si rimane in un certo stato

$$\alpha_D(S_1)(S_2) = \frac{\alpha_c(S_1\ S_2)}{\sum_{S \in \mathbb{S}} \alpha_c(S_1\ S_2)}$$

Normalizzazione

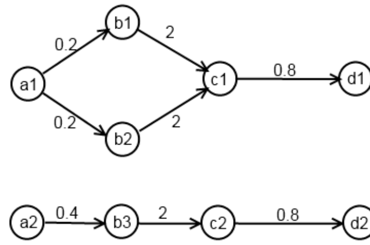


Figura 10.4.: Due Continuous-time Markov Chain bisimilari

10.4.2. Bisimilarità nelle CTMC

A partire dalla bisimilarità per CCS possiamo definire facilmente una nozione analoga di bisimilarità per le CTMC.

Per arrivarci, vediamo la bisimilarità in CCS da un altro punto di vista. Definiamo una funzione $\gamma : S \times Act \times 2^S \to \{\mathbf{true}, \mathbf{false}\}$ che, presi uno stato, un'azione ed un insieme di stati di arrivo, restituisce true se e solo se esiste uno stato fra quelli di arrivo per il quale esiste la transizione data dallo stato di partenza.

$$\gamma(p, \mu, I) = \exists q \in I.\ p \xrightarrow{\mu} q$$

Se dato un agente p ne indichiamo la classe di equivalenza con la notazione $[p]$, abbiamo che

$$p\phi(R)q \Leftrightarrow [p]_{\phi(R)} = [q]_{\phi(R)} \implies$$

$$\forall \mu, I \in R.\ \gamma(p, \mu, I) \Leftrightarrow \gamma(q, \mu, I)$$

In pratica, due agenti sono bisimilari se e solo se entrambi sono nella stessa classe di equivalenza, il che implica che per qualunque classe di equivalenza I e mossa μ la funzione γ restituisce lo stesso valore.

Passando alle CTMC, definiamo la funzione $\gamma_M : S \times 2^S \to \mathbb{R}$, che somma i rate delle mosse dallo stato di partenza all'insieme degli stati di arrivo.

$$\gamma_M(s, I) = \alpha(s)(I) = \sum_{s' \in I} \alpha(s)(s')$$

Notare come questa definizione oltre ad essere qualitativa come quella data per CCS è anche quantitativa perché ritorna un valore ben preciso.
Possiamo introdurre a questo punto una nozione di bisimilarità analoga alla precedente basata sulla definizione di γ_M.

$$s_1 \varphi(R) s_2 \implies \forall I \in R.\ \gamma_M(s_1, I) = \gamma_M(s_2, I)$$

$$\simeq = \bigcup_{R = \varphi(R)} R \qquad \text{CTMC bisimilarity}$$

Avendo un risultato numerico, deve valere l'uguaglianza matematica fra i due valori restituiti. Vediamo un esempio.

Esempio 10.23
Prendiamo la situazione della figura 10.4, e proviamo che le classi di equivalenza sono le seguenti.

$$R = \{a_1, a_2\}, \{b_1, b_2, b_3\}, \{c_1, c_2\}, \{d_1, d_2\}$$

Vogliamo dunque dimostrare che R è una bisimulazione. Per farlo, controlliamo che la proprietà vista sopra è vera.

$$\gamma_M(a_1, \{b_1, b_2, b_2\}) = \gamma_M(a_2, \{b_1, b_2, b_3\}) = 0.4$$

$$\gamma_M(b_1, \{c_1, c_2\}) = \gamma_M(a_2, \{c_1, c_2\}) = 2$$

$$\gamma_M(c_1, \{d_1, d_2\}) = \gamma_M(c_2, \{d_{1,2}\}) = 0.8$$

La relazione effettivamente vale (in tutti i casi non elencati abbiamo che γ_M restituisce zero, non essendoci cammini.

Potremmo fare un discorso simile anche per le DTMC, ma in quel caso avremmo un risultato poco interessante: essendo in ogni nodo la somma dei pesi della stella uscente pari a 1, tutti gli stati risulterebbero bisimilari.

11. Estensioni delle catene di Markov e PEPA

11.1. Azioni e nondeterminismo nelle catene di Markov

Con questo ultimo capitolo vogliamo esplorare le catene di Markov per comprendere come aggiungervi informazioni e renderle più espressive. Per farlo, vogliamo aggiungere due elementi:

- le *azioni* con cui etichettare gli archi della catena
- il *nondeterminismo* a livello di distribuzione: a seconda di quale etichetta scegliamo possamo avere più distribuzioni probabilistiche in uscita.

Nei prossimi paragrafi esploreremo cosa questo significhi.

11.1.1. CTMC con azioni

L'aggiunta di informazioni riguardanti le azioni da intraprendere influenza la nostra funzione di transizione di stato. Partiamo ancora una volta da CCS: abbiamo detto che in CCS potevamo scrivere la α come $S \to L \to \mathscr{P}(S)$ oppure $S \to \mathscr{P}(L \times S)$. Da queste due espressioni equivalenti possiamo generare due versioni delle CTMC con azioni che differiscono leggermente.

$$\begin{array}{ll} \alpha_R : S \to L \to (D(S) + 1) & \text{reactive probabilistic transition system} \\ \alpha_G : S \to (D(L \times S) + 1) & \text{generative probabilistic transition system} \end{array}$$

In un sistema reattivo scelgo prima l'etichetta da seguire, poi trovo una distribuzione di probabilità che mi dice dove posso finire. In un sistema generativo ho la probabilità che mi lega più strettamente: non ho effettivo contollo sul sistema in quanto non posso scegliere l'etichetta da seguire, al contrario di quanto succede nei sistemi reattivi.

In un sistema reattivo dovrà valere $\sum_{s'} \alpha_R\, s\, l\, s' = 1$, mentre in un sistema generativo facciamo la somma anche sulle etichette: $\sum_{(l,s')} \alpha_G\, s\,(l, s') = 1$.

Approfondiremo solamente i sistemi reattivi.

11.1.1.1. Reactive CTMC

Vediamo un esempio.

Esempio 11.1 (Reactive probabilistic transition system)
Vediamo in figura 11.1 un Reactive Probabilistic Transition System. In questo RPTS dallo stato iniziale (in alto) abbiamo due transizioni, una etichettata 1 € ed una etichettata 2 €, che conducono a due distribuzioni probabilistiche che suddividono fra due stati, uno dei quali ci riporta allo stato iniziale con una transizione coffee e l'altro con la transizione cappuccino.

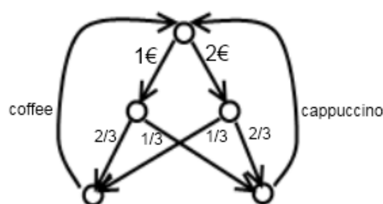


Figura 11.1.: Un automa per un reactive probabilistic transition system

Possiamo definire un concetto di bisimulazione analogo a quello visto per le CTMC, con ovviamente l'aggiunta delle etichette.

$$s_1 \; \varphi(R) \; s_2 \Longrightarrow \alpha(s_1)(l)(I) = \alpha(s_2)(l)(I) \qquad \text{per ogni classe di equivalenza I di R}$$

11.1.1.2. Logica di Larsen-Skou

Introduciamo ora una logica che è una generalizzazione di quella di Hennessy-Milner già vista per il CCS che ci permetterà di definire in maniera alternativa la bisimulazione.

Queste formule hanno una sintassi

$$\varphi ::= \mathbf{true} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \langle a \rangle_q \, \varphi$$

Vediamo di dare la semantica di queste formule. Abbiamo che

$$\begin{array}{lll} s \models \mathbf{true} & & \text{sempre vera} \\ s \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s \models \neg\varphi & \Leftrightarrow & \neg s \models \varphi \\ s \models \langle l \rangle_q \, \varphi & \Leftrightarrow & \alpha \, s \, l \, [\![\varphi]\!] \geq q \\ & & \text{dove } [\![\varphi]\!] = \{s \in S \,|\, s \models \varphi\} \end{array}$$

L'unico operatore differente è il modale, notare che Hennessy-Milner era un caso particolare di questo operatore in cui q era sempre zero.

Esempio 11.2 (Logica di Larsen-Skou)
Vediamo una formula applicata al caso visto nell'immagine 11.1.

É vero che $s_1 \models \langle 1d \rangle \langle coffee \rangle \, \mathbf{true}$*? Intatto cerchiamo di capire cosa significhi. Vogliamo sapere se è vero che, partendo da* s_1 *e muovendosi lungo la transizione* $1d$*, abbiamo più di 1/2 di probabilità di arrivare in uno stato nel quale transitiamo di sicuro per un ramo coffee? Arriviamoci.*

Vogliamo, per definizione, che valga $\alpha \, s_1 \, 1d \, I_1 \geq 1/2$*. Per saperlo, dobbiamo sapere come è fatto* I_1*, che é definito dalla formula* $\langle coffee \rangle \, \mathbf{true}$*. Da questa formula ricaviamo che* $I_1 = \{s \in S \,|\, \alpha s \, \mathbf{coffee} \, I_2 \geq 1\}$*: per calcolare questo insieme dobbiamo ricavare l'insieme* $I_2 = \{s \in S \,|\, s \models \mathbf{true}\}$*. Avremo dunque che* I_2 *sarà costituito da tutti gli stati del RPTS,* I_1 *che conterrà gli stati con le quali transisco con etichetta coffee e probabilità maggiore o uguale ad 1 in uno stato in* I_2*. Quindi abbiamo* $I_1 = \{s_2\}$*.*

La nostra formula è dunque vera: da s_1 *possiamo transire in* s_2 *mediante l'etichetta* $1d$ *con una probabilità maggiore di 2/3.*

Vale anche qua il risultato visto nel caso di CCS.

Teorema 11.3 (Bisimilarità e formule di Larsen-Skou)
Due stati sono bisimilari se e solo se soddisfano le stesse formule di Larsen-Skou.

11.1.2. CTMC con nondeterminismo

Il passo successivo è aggiungere all'interno dei nostri automi il concetto di nondeterminismo. Attenzione: non va confuso il nondeterminismo con la probabilità, in quanto come vedremo gli automi che presenteremo in questa sezione associeranno alla stessa etichetta diverse distribuzioni di probabilità che verranno scelte in maniera nondeterministica.

Presenteremo due tipi di automi: gli automi di Segala e la loro versione semplificata.

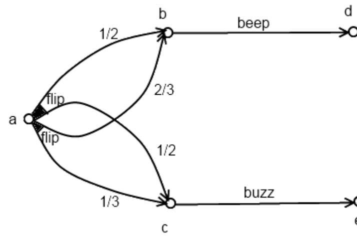


Figura 11.2.: Un Segala automaton

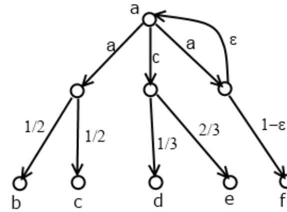


Figura 11.3.: Un Simple-Segala automaton

11.1.2.1. Segala Automaton - Roberto Segala

Un automa di Segala ha una funzione di transizione di questo tipo

$$\alpha_s : S \rightarrow \mathscr{P}(D(L \times S))$$

É una funzione che, dato uno stato, restituisce un insieme di distribuzioni probabilistiche sulle coppie composte da un'etichetta ed uno stato di arrivo. Non c'è più il caso +1 perché nell'insime delle parti c'è l'insieme vuoto. Vediamo un esempio.

Esempio 11.4 (Segala automaton)
Vediamo un esempio di automa di Segala in figura 11.2. Notiamo che dallo stato a ci sono due differenti distribuzioni di probabilità per l'etichetta flip.
Vediamo come è fatta α_S.

$$\alpha_S(a) = \{d_1, d_2\} \quad dove \quad \begin{array}{ll} d_1(flip, b) = \frac{1}{2} & d_1(flip, c) = \frac{1}{2} \\ d_2(flip, b) = \frac{2}{3} & d_2(flip, c) = \frac{1}{3} \end{array}$$

$$\alpha_S(b) = \{d_3\} \quad dove\ d_3(beep, d) = 1$$

$$\alpha_S(c) = \{d_4\} \quad dove\ d_3(buzz, e) = 1$$

$$\alpha_S(d) = \alpha_S(e) = \emptyset$$

11.1.2.2. Simple Segala Automaton

Esiste una semplificazione degli automi di Segala che, similmente ai reactive probabilistic transition systems, tira fuori la scelta dell'etichetta da seguire dalla distribuzione di probabilità. Vediamo la funzione di transizione ed un esempio.
$\alpha_{SS} : S \rightarrow \mathscr{P}(L \times D(S))$

Esempio 11.5 (Simple Segala Automaton)
Vediamo un esempio di Simple Segala Automaton in figura 11.3. Vediamone la funzione α.

$$\alpha_{SS}(a) = \{(a, d_1), (c, d_2), (a, d_3)\} \text{ dove } \begin{array}{l} d_1(b) = d_1(c) = \frac{1}{2} \\ d_2(d) = \frac{2}{3} \quad d_2(e) = \frac{1}{3} \\ d_3(a) = \epsilon \quad d_3(f) = 1 - \epsilon \end{array}$$

$$\alpha_{SS}(b) = \alpha_{SS}(c) = \alpha_{SS}(d) = \alpha_{SS}(e) = \alpha_{SS}(f) = \emptyset$$

11.1.2.3. Altre combinazioni

In letteratura sono state analizzate le possibilità più disparate di combinare queste informazioni: ne riportiamo un paio senza andarle ad analizzare.

Il modello *alternating* prevede due tipi di transizione: la funzione α può alternativamente restituire una distribuzione sugli stati di arrivo oppure un insieme nondeterministico di coppie etichetta-stato, in modo da poter esprimere alternativamente sia la probabilità che il nondeterminismo.

Il modello *bundle* invece restituisce una distribuzione probabilistica di sottoinsiemi delle coppie etichetta-stato: è possibile dunque avere situazioni nelle quali abbiamo una certa probabilità di avere a disposizione certe scelte.

I modelli sono tanti, e la scelta di quale possa servirci dipende fortemente dall'uso che ne vogliamo fare. Vedremo comunque nella prossima sezione che alcuni modelli sono più generali di altri, avendo dunque una maggiore espressività.

11.1.3. Segala come modello più generale

É possibile dimostrare che, fra i modelli presentati, Segala è il più generale. In altre parole, sarà sempre possibile esprimere ad esempio un automa reactive con un automa di Segala mentre non sarà in generale possibile il contrario.

11.1.3.1. Segala vs Generative

Teorema 11.6 (Segala è più espressivo di Generative)
É semplice vedere che è possibile riscrivere ogni situazione del generative con Segala. Infatti

- *Se $\alpha_G(s) = *$, cioè lo stato s é di deadlock, possiamo scrivere l'espressione equivalente $\alpha_S(s) = \emptyset$.*
- *Se $\alpha_G(s)(l_1, s_1) = x$, cioè $\alpha_G(s)$ ci restituisce una distribuzione di probabilità nella quale alla coppia (l_1, s_1) é associato il valore x, facciamo corrispondere una transizione di Segala nella quale $\alpha_S(s)$ ci restituisce un singleton con dentro proprio la distribuzione $\alpha_G(s)$.*

Non è possibile invece esprimere con Generative la situazione in cui $\alpha_S(s)$ ci restituisce più distribuzioni di probabilità.

11.1.3.2. Generative → Segala

$$\frac{\alpha_g(s) = *}{\alpha_s(s) = \emptyset} \qquad \frac{\alpha_g(s_1)(l, s_2) = x}{\alpha_s(s_1) = \{d\} \quad d(l, s_1) = x}$$

11.1.3.3. Reactive → Segala

$$\frac{\alpha_r(s_1)(l)(s_2) = x}{d_l \in \alpha_s(s_1) \quad d_l(l, s_2) = x \quad d_l(l', s) = 0 \quad l' \neq l}$$

11.1.3.4. Simple Segala → Segala

$$\frac{(l,d) \in \alpha_{ss}(s_1) \quad d(s_2) = x}{d_{(l,d)} \in \alpha_s(s_1) \quad d_{(l,d)}(l, s_2) = x \quad d_{(l,d)}(l', s) = 0 \quad l' \neq l}$$

11.1.3.5. Simple Segala vs Generative

11.2. PEPA - Performance Evaluation Process Algebra

Introduciamo un'algebra per descrivere i nostri sistemi di transizione probabilistici, etichettati e non deterministici.

11.2.1. Sintassi di PEPA

$$P ::= (\alpha, \lambda).P \mid P + P \mid P \bowtie P \mid P/L \mid A$$
$$\alpha \in \mathscr{L} = \text{Insieme delle etichette/azioni}$$
$$\lambda \in \mathbb{R} = \textit{rate}$$

PEPA è un derivato del CSP di Tony Hoare piuttosto che del CCS di Milner come il Π-calcolo, e quindi va notato che le azioni hanno un significato differente: in CCS avevamo λ e $\bar{\lambda}$ e questo ci serviva per fare le sincronizzazioni a due, mentre in CSP è possibile fare sincronizzazioni a n agenti e quindi tutti usano la stessa azione α.

$(\alpha, r).P$ operatore di prefisso: l'agente può fare una mossa etichettata α con probabilità r

$+$ stesso significato del CCS, scelta non deterministica

$\bowtie_L$ operatore per la sincronizzazione tramite le $\alpha \in L$

P/L operatore di *hiding*: ogni $\alpha \in L$ diventa una τ

A definizione ricorsiva in cui $A \overset{\text{def}}{=} P$ ed in P è possibile usare A

11.2.2. Semantica operazionale di PEPA

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E} \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \text{ e simmetrica}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \notin L}{E/L \xrightarrow{(\alpha,r)} E'/L} \qquad \frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \in L}{E/L \xrightarrow{(\tau,r)} E'/L}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \notin L}{E \bowtie_L F \xrightarrow{(\alpha,r)} E' \bowtie_L F} \text{ e simmetrica} \qquad \frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F' \quad \alpha \in L}{E \bowtie_L F \xrightarrow{(\alpha,r)} E' \bowtie_L F'}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad A \overset{\text{def}}{=} E}{A \xrightarrow{(\alpha,r)} E'}$$

Parte V.

Appendici

A. Tabelle riassuntive di sintassi e semantica

A.1. IMP

A.1.1. Sintassi

Espressioni aritmetiche (Aexpr) $a :: n|x|a_0 + a_1|a_0 - a_1|a_0 \times a_1$ dove n è un naturale, x è una locazione, a_0, a_1 espressioni aritmetiche.

Espressioni booleane (Bexpr) $b :: v|a_0 = a_1|a_0 \leq a_1|\neg b|b_0 \vee b_1|b_0 \wedge b_1$

Comandi (Com) $c :: \textbf{skip}\,|x := a|c_0; c_1|\,\textbf{if}\; b\; \textbf{then}\; c_0\; \textbf{else}\; c_1|\,\textbf{while}\; b\; \textbf{do}\; c$

A.1.2. Semantica operazionale

A.1.2.1. Regole per Aexpr $< a, \sigma > \to n \in \mathbb{N}$

$$\frac{}{\langle n, \sigma \rangle \to n}\; num \qquad \frac{}{\langle x, \sigma \rangle \to \sigma(x)}\; ide$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 \oplus n_1}\; sum \qquad \frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n_0 \ominus n_1}\; minus \qquad \frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n_0 \otimes n_1}\; mul$$

A.1.2.2. Regole per Bexpr $< b, \sigma > \to v \in \{\textbf{true}, \textbf{false}\}$

$$\frac{}{\langle v, \sigma \rangle \to v}\; bool \qquad \frac{\langle v, \sigma \rangle \to v}{\langle \neg v, \sigma \rangle \to \textbf{not}\; v}\; not$$

$$\frac{\langle b_0, \sigma \rangle \to n_0 \quad \langle b_1, \sigma \rangle \to n_1}{\langle b_0 \wedge b_1, \sigma \rangle \to n_0 \owedge n_1}\; and \qquad \frac{\langle b_0, \sigma \rangle \to n_0 \quad \langle b_1, \sigma \rangle \to n_1}{\langle b_0 \vee b_1, \sigma \rangle \to n_0 \ovee n_1}\; or \qquad \frac{\langle b_0, \sigma \rangle \to n_0 \quad \langle b_1, \sigma \rangle \to n_1}{\langle b_0 = b_1, \sigma \rangle \to n_0 \equiv n_1}\; sum$$

A.1.2.3. Regole per Com $< c, \sigma > \to \sigma' \in \Sigma$

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma}\; skip \qquad \frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[^m/_x]}\; assign \qquad \frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}\; concat$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if}\; b\; \textbf{then}\; c_0\; \textbf{else}\; c_1, \sigma \rangle \to \sigma'}\; if - tt \qquad \frac{\langle b, \sigma \rangle \to \textbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if}\; b\; \textbf{then}\; c_0\; \textbf{else}\; c_1, \sigma \rangle \to \sigma'}\; if - ff$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \textbf{while}\; b\; \textbf{do}\; c, \sigma'' \rangle \to \sigma'}{\langle \textbf{while}\; b\; \textbf{do}\; c, \sigma \rangle \to \sigma'}\; while - tt \qquad \frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \textbf{while}\; b\; \textbf{do}\; c, \sigma \rangle \to \sigma}\; while - ff$$

A.1.3. Semantica denotazionale

A.1.3.1. $\mathscr{A} : Aexpr \to \Sigma \to \mathbb{N}$

$$\mathscr{A}\,[\![n]\!]\,\sigma = n \qquad \mathscr{A}\,[\![x]\!]\,\sigma = \sigma x$$

$$\mathscr{A}\,[\![a_0 + a_1]\!]\,\sigma = \mathscr{A}\,[\![a_0]\!]\,\sigma \oplus \mathscr{A}\,[\![a_1]\!]\,\sigma$$

$$\mathscr{A}\,[\![a_0 - a_1]\!]\,\sigma = \mathscr{A}\,[\![a_0]\!]\,\sigma \ominus \mathscr{A}\,[\![a_1]\!]\,\sigma$$

$$\mathscr{A}\,[\![a_0 \times a_1]\!]\,\sigma = \mathscr{A}\,[\![a_0]\!]\,\sigma \otimes \mathscr{A}\,[\![a_1]\!]\,\sigma$$

A.1.3.2. $\mathscr{B} : Bexpr \to \Sigma \to \mathbb{B}$

$$\mathscr{B}\,[\![v]\!]\,\sigma = v \qquad \mathscr{B}\,[\![\neg v]\!]\,\sigma = \mathbf{not}\,[\![v]\!]\,\sigma$$

$$\mathscr{B}\,[\![b_0 = b_1]\!]\,\sigma = \mathscr{B}\,[\![b_0]\!]\,\sigma \equiv \mathscr{B}\,[\![b_1]\!]\,\sigma$$

$$\mathscr{B}\,[\![b_0 \wedge b_1]\!]\,\sigma = \mathscr{B}\,[\![b_0]\!]\,\sigma \owedge \mathscr{B}\,[\![b_1]\!]\,\sigma$$

$$\mathscr{B}\,[\![b_0 \vee b_1]\!]\,\sigma = \mathscr{B}\,[\![b_0]\!]\,\sigma \ovee \mathscr{B}\,[\![b_1]\!]\,\sigma$$

A.1.3.3. $\mathscr{C} : Com \to \Sigma \to \Sigma_\perp$

$$\mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma$$

$$\mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma\left[{}^{\mathscr{A}[\![a]\!]}/_{x}\right]$$

$$\mathscr{C}\,[\![c_1; c_2]\!]\,\sigma = \mathscr{C}\,[\![c_2]\!]^*\,(\mathscr{C}\,[\![c_1]\!]\,\sigma)$$

$$\mathscr{C}\,[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\,\sigma = \mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c_1]\!]\,\sigma, \mathscr{C}\,[\![c_2]\!]\,\sigma$$

$$\mathscr{C}\,[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \mathbf{fix}\,\Gamma, \text{ dove } \Gamma = \lambda\varphi.\lambda\sigma.\mathscr{B}\,[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}\,[\![c]\!]\,\sigma), \sigma$$

A.2. HOFL

A.2.1. Sintassi

Tipi $\tau := \quad int \quad | \quad \tau_1 \times \tau_2 \quad | \quad \tau_1 \rightarrow \tau_2$

Termini

$t ::$	$x \quad \vert$	variabile
	$n \quad \vert$	costante
	$t_1 + t_2 \quad \vert \quad t_1 - t_2 \quad \vert \quad t_1 \times t_2 \quad \vert$	operazioni aritmetiche
	$\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \quad \vert$	condizionale
	$(t_1, t_2) \quad \vert \quad \textbf{fst}(t) \quad \vert \quad \textbf{snd}(t) \quad \vert$	operazioni su coppie
	$\lambda x.t \quad \vert \quad (t_1\ t_2) \quad \vert$	definizione e applicazione di funzione
	$\textbf{rec}\ x.t$	funzioni ricorsive

A.2.2. Regole di inferenza dei tipi

Variabili $x : type(x) = \hat{x}$

Operazioni

$$n : int \qquad \frac{t_1 : int \quad t_2 : int}{t_1 \text{ op } t_2 : int} \qquad \frac{t_0 : int \quad t_1 : \tau \quad t_2 : \tau}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \tau} \qquad \text{con op = +, -, x}$$

Prodotti

$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1, t_2) : (\tau_1 * \tau_2)} \qquad \frac{t : \tau_1 * \tau_2}{\textbf{fst}(t) : \tau_1} \qquad \frac{t : \tau_1 * \tau_2}{\textbf{snd}(t) : \tau_2}$$

Funzioni

$$\frac{x : \tau_1 \quad t : \tau_2}{\lambda x.t : \tau_1 \rightarrow \tau_2} \qquad \frac{t_1 : \tau_1 \rightarrow \tau_2 \qquad t_2 : \tau_1}{(t_1\ t_2) : \tau_2}$$

Rec

$$\frac{x : \tau \quad t : \tau}{\textbf{rec}\ x.t : \tau}$$

A.2.3. Semantica operazionale

A.2.3.1. Forme canoniche

$$\frac{}{n \in C_{int}} \qquad \frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1, t_2 : \text{ closed}}{(t_1, t_2) \in C_{\tau_1 * \tau_2}} \qquad \frac{\lambda x.t : \tau_1 \to \tau_2 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_1 \to \tau_2}}$$

A.2.3.2. Regole di inferenza

$$\frac{}{c \to c}$$

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \underline{op} n_2} \qquad \frac{t_0 \to 0 \quad t_1 \to c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_1} \qquad \frac{t_0 \to n \quad n \neq 0 \quad t_2 \to c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_2}$$

$$\frac{t \to (t_1, t_2) \quad t_1 \to c_1}{\textbf{fst}(t) \to c_1} \qquad \frac{t \to (t_1, t_2) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_2}$$

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c} (LAZY) \qquad \frac{t_1 \to \lambda x.t_1' \quad t_2 \to c_2 \quad t_1'[^{c_2}/_x] \to c}{(t_1 \quad t_2) \to c} (EAGER)$$

$$\frac{t[^{\textbf{rec}\, x.t}/_x] \to c}{\textbf{rec}\, x.t \to c}$$

A.2.4. Semantica denotazionale

Date

$$\underline{op}_\perp(x_1, x_2) = \begin{cases} \lfloor x_1 \underline{op} x_2 \rfloor & \text{se } \lfloor x_1 \rfloor \neq \perp_{\mathbb{N}_\perp} \wedge \lfloor x_2 \rfloor \neq \perp_{\mathbb{N}_\perp} \\ \perp_{\mathbb{N}_\perp} & \text{altrimenti} \end{cases} \qquad Cond_\tau(z_0, z_1, z_2) = \begin{cases} z_1 & \text{se } z_0 = \lfloor 0 \rfloor \\ z_2 & \text{se } z_0 = \lfloor 0 \rfloor \quad z \neq 0 \\ \perp_\tau & \text{se } z_0 = \perp_{\mathbb{N}} \end{cases}$$

ho

$$[\![n]\!]\rho = \lfloor n \rfloor$$

$$[\![x]\!]\rho = \rho x$$

$$[\![t_1 \textbf{ op } t_2]\!]\rho = [\![t_1]\!]\rho \; \underline{op}_\perp \; [\![t_2]\!]\rho$$

$$[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]\rho = Cond\,([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

$$[\![(t_1, t_2)]\!]\rho = \lfloor ([\![t_1]\!]\rho, [\![t_2]\!]\rho) \rfloor$$

$$[\![fst(t)]\!]\rho = \textbf{let}\, v \Leftarrow [\![t]\!]\rho. \quad \pi_1 v$$

$$[\![snd(t)]\!]\rho = \textbf{let}\, v \Leftarrow [\![t]\!]\rho. \quad \pi_2 v$$

$$[\![\lambda x.t]\!]\rho = \left\lfloor \lambda d.\, [\![t]\!]\rho[^d/_x] \right\rfloor$$

$$[\![(t_1 t_2)]\!] = \textbf{let}\, \varphi \Leftarrow [\![t_1]\!]\rho.\varphi([\![t_2]\!]\rho)$$

$$[\![rec x.t]\!]\rho = \textbf{fix}\, \lambda d.\, [\![t]\!]\rho[^d/_x]$$

A.3. CCS

A.3.1. Sintassi

$\Delta ::= \alpha, \beta, \gamma, \delta$	Azioni: generico nome α
$\Lambda ::= \Delta \mid \bar{\Delta}$	Azioni visibili: generico nome λ
$\tau \notin \Lambda$	Azione invisibile
	$\Lambda \cup \{\tau\}$ generico nome μ

$$p ::= x \quad | \quad nil \quad | \quad \mu.p \quad | \quad \underbrace{p\backslash\alpha}_{\text{p ristretto a } \alpha} \quad | \quad \underbrace{p[\Phi]}_{\text{permutazione di } \Delta} \quad | \quad p+p \quad | \quad p \mid p \quad | \quad \mathbf{rec}\, x.p$$

A.3.2. Semantica operazionale

$$\text{(Act)} \quad \mu.p \xrightarrow{\mu} p \qquad\qquad \text{(Res)} \quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha} \quad \mu \neq \alpha, \bar{\alpha} \tag{A.1}$$

$$\text{(Rel)} \quad \frac{p \xrightarrow{\mu} q}{p[\Phi] \xrightarrow{\Phi(\mu)} q[\Phi]} \qquad\qquad \text{(Sum)} \quad \frac{p \xrightarrow{\mu} q}{p + r \xrightarrow{\mu} q} \qquad \frac{p \xrightarrow{\mu} q}{r + p \xrightarrow{\mu} q} \tag{A.2}$$

$$\text{(Com)} \quad \frac{p \xrightarrow{\mu} q}{p|r \xrightarrow{\mu} q|r} \qquad \frac{p \xrightarrow{\mu} q}{r|p \xrightarrow{\mu} r|q} \qquad \frac{p_1 \xrightarrow{\lambda} q_1 \quad p_2 \xrightarrow{\lambda} q_2}{p_1|p_2 \xrightarrow{\tau} q_1|q_2} \tag{A.3}$$

$$\text{(Rec)} \quad \frac{p[\mathbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\mathbf{rec}\, x.p \xrightarrow{\mu} q} \tag{A.4}$$

A.3.3. Strong Bisimilarity

$$\simeq \quad \begin{array}{l} p \xrightarrow{\mu} p' \Longrightarrow \exists q'.\, q \xrightarrow{\mu} q' \quad \text{and} \quad p' \simeq q' \quad \wedge \\ q \xrightarrow{\mu} q' \Longrightarrow \exists p'.\, p \xrightarrow{\mu} p' \quad \text{and} \quad p' \simeq q' \end{array}$$

A.3.4. Weak Observational equivalence

$$p \overset{\epsilon}{\Longrightarrow} q \quad sse \quad p \xrightarrow{\tau} \ldots \xrightarrow{\tau} q \quad \text{o} \quad p = q$$

$$p \overset{\lambda}{\Longrightarrow} q \quad sse \quad p \overset{\epsilon}{\Rightarrow} p' \xrightarrow{\lambda} q' \overset{\epsilon}{\Longrightarrow} q$$

$$\begin{array}{lllllll} p \approx q & sse & p \overset{\epsilon}{\Longrightarrow} p' & implica & q \overset{\epsilon}{\Longrightarrow} q' & e & p' \approx q' \\ & sse & p \overset{\lambda}{\Longrightarrow} p' & implica & q \overset{\lambda}{\Longrightarrow} q' & e & p' \approx q' \end{array}$$

e viceversa

A.3.5. Weak Observational Congruence

$$\begin{array}{lllllll} p \approx_c q & sse & p \xrightarrow{\tau} p' & implica & q \xrightarrow{\tau} \overset{\epsilon}{\Longrightarrow} q' & e & p' \approx q' \\ & sse & p \xrightarrow{\lambda} p' & implica & q \overset{\lambda}{\Longrightarrow} q' & e & p' \approx q' \end{array}$$

e viceversa

A.3.6. Dynamic congruence

$$\begin{array}{llll} p \approx_D q & sse \quad p \xrightarrow{\tau} p' \text{ implies } q \xrightarrow{\tau}\overset{\epsilon}{\Longrightarrow} q' & \text{and} & p' \approx_D q' \\ & sse \quad p \xrightarrow{\lambda} p' \text{ implies } q \xrightarrow{\lambda}\overset{\epsilon}{\Longrightarrow} q' & \text{and} & p' \approx_D q' \\ & & \text{e viceversa} & \end{array}$$

A.4. Π-calcolo

A.4.1. Sintassi

$$P ::= \mathbf{nil} \mid \alpha.p \mid [x = y]p \mid p + p \mid p \mid p \mid (y)p \mid !p$$
$$\alpha ::= \tau \mid x(y) \mid \overline{x}\, y$$

A.5. Semantica operazionale

$$\frac{}{\tau p \overset{\tau}{\to} p} \qquad \frac{}{\overline{x}\, y.p \xrightarrow{\overline{x}\, y} p} \qquad \frac{}{x(y).p \xrightarrow{x(w)} p[^w/_y]}\; w \notin fn((y)p)$$

$$\frac{p \overset{\alpha}{\to} p'}{p + q \overset{\alpha}{\to} p'} \qquad \frac{q \overset{\alpha}{\to} q'}{p + q \overset{\alpha}{\to} q'} \qquad \frac{p \overset{\alpha}{\to} p'}{[x = x]p \overset{\alpha}{\to} p'} \qquad \frac{p|!p \overset{\alpha}{\to} p'}{!p \overset{\alpha}{\to} p'}$$

$$\frac{p \overset{\alpha}{\to} p'}{p|q \overset{\alpha}{\to} p'|q}\; bn(\alpha) \cap fn(q) = \emptyset \qquad \frac{q \overset{\alpha}{\to} q'}{p|q \overset{\alpha}{\to} p|q'}\; bn(\alpha) \cap fn(p) = \emptyset \qquad \frac{p \xrightarrow{\overline{x}\, z} p' \quad q \xrightarrow{x(y)} q'}{p|q \overset{\tau}{\to} p'|(q'[^z/_y])}$$

$$\frac{p \overset{\alpha}{\to} p'}{(y)p \overset{\alpha}{\to} (y)p'}\; y \notin n(\alpha) \qquad \frac{p \xrightarrow{\overline{x}\, y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'[^w/_y]}\; y \neq x \quad w \notin fn((y)p) \qquad \frac{p \xrightarrow{\overline{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \overset{\tau}{\to} (w)(p'|q')}$$

A.5.0.1. Bisimilarità ground

$$p \overset{\circ}{\sim} q \Leftrightarrow p \overset{\alpha}{\to} p' \quad \alpha \neq x(y) \quad \wedge \quad bn(\alpha) \notin fn(q) \quad \Rightarrow q \overset{\alpha}{\to} q' \quad \wedge \quad p' \overset{\circ}{\sim} q' \quad \text{e viceversa}$$

$$p \overset{\circ}{\sim}_E q \Leftrightarrow p \xrightarrow{x(y)} p' \quad y \notin fn(q) \quad \Rightarrow \quad \forall w.\exists q'.q \xrightarrow{x(y)} q' \quad p'[^w/_y] \overset{\circ}{\sim}_E q'[^w/_y] \text{ e viceversa}$$

$$p \overset{\circ}{\sim}_L q \Leftrightarrow p \xrightarrow{x(y)} p' \quad y \notin fn(q) \quad \Rightarrow \quad \exists q'.\forall w.q \xrightarrow{x(y)} q' \quad p'[^w/_y] \overset{\circ}{\sim}_L q'[^w/_y] \text{ e viceversa}$$

A.5.0.2. Bisimilarità non ground

$$p \sim_E q \Leftrightarrow \forall \sigma.p\sigma \overset{\circ}{\sim}_E q\sigma$$

$$p \sim_L q \Leftrightarrow \forall \sigma.p\sigma \overset{\circ}{\sim}_L q\sigma$$

A.6. Modelli probabilistici

A.6.1. DTMC

$$\alpha : S \to (D(S) + 1) \qquad \text{Unione disgiunta con singleton}$$
$$\sum_i \alpha(s)(s_i) = 1 \qquad \text{con } \alpha(s) \in D(S)$$

A.6.2. CTMC

$$\alpha : S \to \mathscr{P}(\mathbb{R} \times S)$$
$$\alpha : (S \times S) \to \mathbb{R}$$
$$\gamma_M(s, I) = \alpha(s)(I) = \sum_{s' \in I} \alpha(s)(s')$$

$$s_1 \varphi(R) s_2 \Longrightarrow \forall I \in R.\ \gamma_M(s_1, I) = \gamma_M(s_2, I)$$

A.6.3. Reactive

$$\alpha_R : S \to L \to (D(S) + 1) \qquad \text{reactive probabilistic transition system}$$
$$\sum_{s'} \alpha_R\, s\, l\, s' = 1$$

A.6.4. Generative

$$\alpha_G : S \to (D(L \times S) + 1) \qquad \text{generative probabilistic transition system}$$
$$\sum_{(l,s')} \alpha_G\, s\, (l, s') = 1$$

$$s_1\ \varphi(R)\ s_2 \Longrightarrow \alpha(s_1)\,(l)\,(I) = \alpha(s_2)\,(l)\,(I) \qquad \text{per ogni classe di equivalenza I di R}$$

A.6.5. Segala

$$\alpha_s : S \to \mathscr{P}(D(L \times S))$$

A.6.6. Simple Segala

$$\alpha_{SS} : S \to \mathscr{P}(L \times D(S))$$

A.7. PEPA

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E} \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \quad \text{e simmetrica}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \notin L}{E/L \xrightarrow{(\alpha,r)} E'/L} \qquad \frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \in L}{E/L \xrightarrow{(\tau,r)} E'/L}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad \alpha \notin L}{E \bowtie_L F \xrightarrow{(\alpha,r)} E' \bowtie_L F} \quad \text{e simmetrica} \qquad \frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F' \quad \alpha \in L}{E \bowtie_L F \xrightarrow{(\alpha,r)} E' \bowtie_L F'}$$

$$\frac{E \xrightarrow{(\alpha,r)} E' \quad A \stackrel{\text{def}}{=} E}{A \xrightarrow{(\alpha,r)} E'}$$

B. Esercizi

B.0.1. Esercizi prima parte

Esercizio 1 Si definisca per ricorsione ben fondata la funzione

$$Vars : \mathbf{Com} \longrightarrow P_{Loc}$$

che, dato un comando, restituisce l'insieme delle locazioni che appaiono alla sinistra di qualche assegnamento. Si dimostri che $\forall c \in \mathbf{Com}, \forall \sigma, \sigma' \in \Sigma$

$$\langle c, \sigma \rangle \to \sigma' \quad \text{implica} \quad \forall x \notin Vars(c).\sigma(x) = \sigma'(x)$$

Svolgimento
Ricorsione Stutturale Le varabili che non sono in c non vengono cambiate

$$\langle c, \sigma \rangle \to \sigma' \implies \forall x \notin Vars(c).\sigma(x) = \sigma(x)$$

Analizziamo tutti i comandi:

- $Vars(\mathbf{skip}) = \emptyset$
- $Vars(x := a) = \{x\}$
- $Vars(c_0; c_1) = Vars(\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1) = Vars(c_0) \cup Vars(c_1)$
- $Vars(\mathbf{while}\, b\, \mathbf{do}\, c) = Vars(c)$

Induzione sulle regole:

$$P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin Vars(c).\ \sigma(x) = \sigma'(x)$$

Prima regola di inferenza (skip)

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$P(\langle \mathbf{skip}, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin \emptyset.\ \sigma(x) = \sigma(x)$$

Seconda regola di inferenza (assegnamento)

$$\frac{\langle a, \sigma \rangle \to n}{\langle x := a, \sigma \rangle \to \sigma[^n/_x]}$$

$$P(\langle x := a, \sigma \rangle \to \sigma[^n/_x]) \stackrel{\text{def}}{=} \forall y \neq x.\ \sigma(y) = \sigma[^n/_x](y)$$

Terza regola di inferenza (concatenamento)

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Grazie all'induzione strutturale, possiamo assumere:

$$P(\langle c_0, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \forall x \notin Vars(c_0).\ \sigma(x) = \sigma''(x)$$

$$P(\langle c_1, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin Vars(c_1).\ \sigma''(x) = \sigma'(x)$$

Dobbiamo dimostrare

$$P(\langle c_0; c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin Vars(c_0; c_1).\ \sigma(x) \stackrel{?}{=} \sigma'(x)$$

Posso assumere

$$x \notin Vars(c_0; c_1) \quad x \notin Vars(c_0) \cup Vars(c_1)$$

Per la proprietà dell'unione abbiamo

$$x \notin Vars(c_0) \quad x \notin Vars(c_1)$$

A questo punto posso dire

$$\sigma(x) = \sigma''(x) \quad \sigma''(x) = \sigma'(x) \quad \sigma(x) = \sigma'(x)$$

che era la cosa che dovevamo dimostrare.

Quarta regola di inferenza (if-then-else)

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

Assumiamo come ipotesi dell'induzione strutturale:

$$P(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin Vars(c_0).\ \sigma(x) = \sigma'(x)$$

Dobbiamo dunque dimostrare:

$$P(\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall x \notin Vars(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1).\ \sigma(x) = \sigma'(x)$$

$$= \forall x \notin Vars(c_0) \cup Vars(c_1).\ \sigma(x) = \sigma'(x)$$

Assumiamo la premessa di ciò che dobbiamo dimostrare, ovvero $x \notin Vars(c_0) \cup Vars(c_1)$. Questo implica che $x \notin Vars(c_0)$. Si noti che, per l'ipotesi induttiva, questo implica che $\forall x \notin Vars(c_0) \cup Vars(c_1).\ \sigma(x) = \sigma'(x)$, che è ciò che volevamo dimostrare.

Il caso **false** è analogo

Sesta regola di inferenza (While)

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma}$$

$$P(\langle \textbf{while}\, b\, \textbf{do}\, c\rangle \to \sigma) \overset{\text{def}}{=} \forall x \notin Vars(\textbf{while}\, b\, \textbf{do}\, c).\ \sigma(x) = \sigma(x)$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'}$$

Assumo

$$P(\langle c_0, \sigma\rangle \to \sigma'') \overset{\text{def}}{=} \forall x \notin Vars(c).\ \sigma(x) = \sigma''(x)$$

$$P(\langle w, \sigma\rangle \to \sigma'') \overset{\text{def}}{=} \forall x \notin Vars(w).\ \sigma(x)'' = \sigma'(x)$$

Dimostro

$$P(\langle w, \sigma\rangle \to \sigma') \overset{\text{def}}{=} \forall x \notin Vars(w).\ \sigma(x) = \sigma'(x)$$

$x \notin Vars(w) = x \notin Vars(c)$ che è vero per ipotesi $\implies$ posso usare l'ipotesi è vedo che $\sigma(x) = \sigma'(x)$

Esercizio 2 Si indichi con w il comando del linguaggio IMP

$$\textbf{while}\, x \neq 0\, \textbf{do}(x := x - 1; y = y + 1)$$

Si provi per induzione sulle regole che, dati comunque gli stati σ e σ',

$$\langle w, \sigma\rangle \to \sigma' \implies \sigma(x) \geq 0 \land \sigma' = \sigma[{}^{\sigma(x)+\sigma(y)}/_{y,0/x}]$$

Svolgimento

Prima Regola

$$\frac{\langle x \neq 0, \sigma\rangle \to \textbf{false}}{\langle w, \sigma\rangle \to \sigma}$$

allora $\sigma(x) = 0$ Il primo pezzettino del and è soddisfatto

$$\sigma[{}^{0+\sigma(y)}/_{y}, {}^{0}/_{x}] = \sigma$$

Seconda Regola

$$\frac{\langle x \neq 0, \sigma\rangle \to \textbf{true} \quad \langle x := x - 1; y := y + 1;\ , \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'}$$

Assumiamo l'ipotesi induttiva, otteniamo quindi:

$$\langle w, \sigma''\rangle \to \sigma' \implies \sigma''(x) \geq 0 \land \sigma' = \sigma''[{}^{\sigma''(x)+\sigma''(y)}/_{y,0/x}]$$

Bisogna dimostrare

- $\sigma(x) \geq 0$
- $\sigma' = \sigma[{}^{\sigma(x)+\sigma(y)}/_{y,0/x}]$

Dimostriamo 1

$$\sigma''(y) = \sigma(y) + 1 \qquad \wedge \qquad \sigma''(x) = \sigma(x) - 1 \equiv \sigma(x) = \underbrace{\sigma''(x)}_{\geq 0 \text{ per ip. ind.}} +1 \geq 0$$

Dimostriamo 2
Sostituiamo

$$\sigma'' = \sigma[^{\sigma(x)-1}/_{x,\sigma(y)+1/y}]$$

$$\begin{aligned}
\sigma' &= \sigma''[^{\sigma''(x)+\sigma''(y)}/_{y,0/x}] = \sigma[^{\sigma(x)-1}/_{x,\sigma(y)+1/y}][^{\sigma''(x)+\sigma''(y)}/_{y,0/x}] \\
&= \underbrace{\sigma[^{\sigma(x)-1}/_{x,\sigma(y)+1/y}]}_{\sigma''}\underbrace{[^{\sigma(x)-1+\sigma(y)+1}/_{y,0/x}]}_{\text{sovrascrittura}} \\
&= \sigma[^{\sigma(x)-1+\sigma(y)+1}/_{y,0/x}] \\
&= \sigma[^{\sigma(x)+\sigma(y)}/_{y,0/x}] \\
&\square
\end{aligned}$$

Terzo esercizio Sia R una relazione su un insieme A, ovvero $R \subseteq A \times A$. Si consideri la relazione R^+, detta *chiusura transitiva* di R, definita dalle seguenti regole:

$$R^+ = \quad \frac{xRy}{xR^+y} \quad \frac{xR^+y \quad yR^+z}{xR^+z}$$

- si dimostri che per ogni x e y

$$xR^+y \quad \Leftrightarrow \exists k > 0.\ \exists z_0, \cdots, z_k.\ .x = z_0 \vee z_0Rz_1 \vee \cdots \vee z_{k-1}Rz_k \vee z_k = y$$

  (Suggerimento: si dimostri l'implicazione $\Rightarrow$ per induzione sulle regole e l'implicazione $\Leftarrow$ per induzione sulla lunghezza k della catena).

- Quali regole definiscono una relazione R' che soddisfi invece

$$xR'y \quad \Leftrightarrow \quad \exists k \geq .\exists z_0, \cdots, z_k.x = z_0 \vee z_0Rz_1 \vee \cdots \vee z_{k-1}Rz_k = y$$

Dimostrazione primo punto

Dimostrazione $\Longrightarrow$

$$xRy(k = 1)\text{ho trovato due z che.. allora } x = y$$

Seconda parte sulla struttura

$$\exists k' > 0 \quad \exists z'_0, .., x'k \quad x = x'_0 \quad z'_0Rz'_1...z'_{k-1}Rz'_k \quad z'_k = y$$

$$\exists k'' > 0 \quad \exists z''_0, .., x''k \quad x = x'_0 \quad z''_0Rz''_1...z''_{k-1}Rz''_k \quad z''_k = z$$

Concateno: $k = k' + k''$

Dimostrazione $\Longleftarrow$ non so cosa valga k.

$$\forall k > 0(\exists z_0...z_k \quad x = z_0Rz_1...z_{n-1}Rz_n = y \Longrightarrow xR^+y)$$

Assomiglia all'induzioni sui naturali ma non esattamente
Caso $k = 1$
Deve succedere che $xRy \Longrightarrow xR^+y$ OK(lo dice la regola di inferenza)

Vediamo per k

$$\exists z'_0, ..., z'_{k+1} \quad x = z'_0Rz'_1...z'_{k-1}Rz..... \Longrightarrow xR^+y?$$

Rompo:

$$\exists z'_0...z'_k \quad x = z'_0Rz'_1...z'_{k-1}Rz'_k$$

Questa suppioniamo che per ipotesi induttiva che da : $xR^+z'_k$

$$z'_kRz'_{k+1} = y$$

questa per il caso $k = 1$ gia dimostrato da z'_kRy.
Allora

$$\dfrac{xR^+z'_k \qquad \dfrac{x'_kRy}{x'_kR^+y}}{xR^+y}$$

Dimostrazione secondo punto Invece di avere $k > 0$ immaginiamo di avere $k \geq 0$ Qual'è la relazione che soddisfa? x e y devono essere riflessive. Se aggiungiamo xR^*y OK. Bisogna fare un'altra provetta.

Esercizio 4 Se non c'è while si usa indizione struttura e si fa vedere che esiste sempre σ'.

Esercizio 5

$$\dfrac{}{(n, m) \to m} \qquad \dfrac{(n, m) \to k}{(m, n) \to k}\ (m < n) \qquad \dfrac{(m - n, n) \to k}{(n, m) \to k}\ (m < n)$$

Dimostrare

$$(m, n) \to \quad k \text{ è il MCD di m e n}$$

Assiumiamo k mcd di $m - n, m \Longrightarrow k$ MCD di m e n

Dimostriamo un'altra proprietà:

$$k \text{ divisore di } m - n, n \Leftrightarrow k \text{ divisore di } m \text{ e } n$$

$\Longrightarrow$

$$k \cdot h_1 = m - n \quad k \cdot h_2 = n$$

Otteniamo

$$k(k_1 + k_2) = m$$

□

$\Longleftarrow$

$$k \cdot h_1 = m \quad k \cdot h_2 = n \quad k_1 > k_2$$

Otteniamo

$$k(h_1 - h_2) = m - n$$

□

B.0.2. Esercizi 23 Marzo 2010

Esercizio 1 TODO: Da rivedere Si introduca il comando

$$\textbf{reapeat}\, n\, \textbf{times}\, c$$

con n naturale, al posto del comando **while** in **IMP**

La semantica denotazionale di tale comando è

$$\mathscr{C}[\![\textbf{repeat}\, n\, \textbf{times}\, c]\!]\sigma = (\mathscr{C}[\![c]\!]^n)\sigma$$

con

$$(\mathscr{C}[\![c]\!])^0\sigma = \sigma \qquad (\mathscr{C}[\![c]\!])^{n+1}\sigma = ((\mathscr{C}[\![c]\!])^n)^*(\mathscr{C}[\![c]\!]\sigma)$$

a) Si definisca la semantica operazionale del nuovo comando;

b) Si modifichi la prova di equivalenza fra semantica operazionale e denotazionale per tener conto del nuovo comando;

c) si dimostri che l'esecuzione di ciascun comando nel linguaggio modificato termina.

Svolgimento Sostituire il costrutto while con repeat impoverisce sicuramente il linguaggio [TODO??? c'erano delle considerazioni sulla touring-equivalenza etc chiedere a qualcuno che ha fatto CC].

a) Semantica operazionale

$$\langle \textbf{repeat}\, 0\, \textbf{times}\, c, \sigma\rangle \to \sigma$$

$$\frac{\langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{repeat}\, n-1\, \textbf{times}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{repeat}\, n\, \textbf{times}\, c, \sigma\rangle \to \sigma'} \quad n > 0$$

b) Dimostriamo l'equivalenza delle due semantiche, mostrando la loro implicazione nei due versi.

b.1) Dimostriamo per induzione sulle regole che l'operazionale definita è equivalente alla denotazionale data.

$$P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$$

- **Prima regola**

$$\langle \textbf{repeat}\, 0\, \textbf{times}\, c, \sigma\rangle \to \sigma \qquad \mathscr{C}[\![\textbf{repeat}\, 0\, \textbf{times}\, c]\!]\sigma \stackrel{?}{=} \sigma$$

questa dimostrazione si deriva immediatamente dalla definizione di $(\mathscr{C}[\![c]\!])^0$.

- **Seconda regola**

$$\frac{\langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{repeat}\, n-1\, \textbf{times}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{repeat}\, n\, \textbf{times}\, c, \sigma\rangle \to \sigma'} \quad n > 0$$

$$P(\langle \textbf{repeat}\, n\, \textbf{times}\, c, \sigma\rangle \to \sigma')) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{repeat}\, n\, \textbf{times}\, c]\!]\sigma \stackrel{?}{=} \sigma'$$

$$(\mathscr{C}[\![c]\!]^{n-1})^* \underbrace{(\mathscr{C}[\![c]\!]\sigma)}_{\sigma''} \stackrel{?}{=} \sigma'$$

Per ipotesi induttiva sulle regole sappiamo che $\langle c, \sigma\rangle \to \sigma''$ e quindi che

$$\mathscr{C}[\![c]\!]\sigma = \sigma''$$

Quindi la stella non serve

$$\mathscr{C}[\![c]\!]^{n-1}\sigma'' \stackrel{?}{=} \sigma'$$

L'induzione inoltre ci permette di ipotizzare che la proprietà che vogliamo dimostrare valga per le premesse, ossia:

$$P(\langle \textbf{repeat}\, n-1\, \textbf{times}\, c, \sigma'' \rangle \to \sigma')) \stackrel{def}{=} \mathscr{C}[\![\textbf{repeat}\, n-1\, \textbf{times}\, c]\!]\sigma'' = \sigma'$$

Possiamo quindi affermare che:

$$\mathscr{C}[\![c]\!]^{n-1}\sigma'' = \sigma'$$

b.2) Dimostriamo che la semantica denotazionale implica l'operazionale.
Essendoci un solo costrutto, bisogna mostrare la proprietà per ogni n quindi utilizziamo l'induzione matematica

$$P(\textbf{repeat}\, n\, \textbf{times}\, c) \stackrel{def}{=} \mathscr{C}[\![\textbf{repeat}\, n\, \textbf{times}\, c]\!]\sigma = \sigma' \implies \langle \textbf{repeat}\, n\, \textbf{times}\, c, \sigma \rangle \to \sigma'$$

Caso $n = 0$

$$P(\textbf{repeat}\, 0\, \textbf{times}\, c) \stackrel{def}{=} \mathscr{C}[\![\textbf{repeat}\, 0\, \textbf{times}\, c]\!]\sigma = \sigma' \implies \langle \textbf{repeat}\, 0\, \textbf{times}\, c, \sigma \rangle \to \sigma'$$

vera per la definizione della semantica denotazionale.

Caso $n \to n+1$

$$P(\textbf{repeat}\, n+1\, \textbf{times}\, c) \stackrel{def}{=} \mathscr{C}[\![\textbf{repeat}\, n+1\, \textbf{times}\, c]\!]\sigma = \sigma' \implies \langle \textbf{repeat}\, n+1\, \textbf{times}\, c, \sigma \rangle \stackrel{?}{\to} \sigma'$$

Assumiamo la premessa

$$(\mathscr{C}[\![c]\!])^n)(\mathscr{C}[\![c]\!]\sigma) = \sigma'$$

Dato che il comando termina in σ', possiamo eliminare la stella e sappiamo che $\mathscr{C}[\![c]\!])\sigma = \sigma''$ da cui $\langle c, \sigma \rangle \to \sigma''$.

Inoltre da

$$(\mathscr{C}[\![c]\!])^n)\sigma'' = \sigma'$$

abbiamo

$$\langle \textbf{repeat}\, n\, \textbf{times}\, c, \sigma'' \rangle \to \sigma'$$

Ora abbiamo le due ipotesi induttive induttive della regola operazionale

$$\langle \textbf{repeat}\, n+1\, \textbf{times}\, c, \sigma \rangle \to \sigma'$$

che risulta dunque verificata.

c) –PEER REVIEW–
Le prove per tutti gli altri costrutti si risolvono semplicemente assumendo che le premesse teminino e mostrando che facendo un passo in più termina anche il teorema.
Stesso vale per **repeat**, le premesse ci dicono che il comando c termina e che **repeat n** termina in σ', da cui deduciamo che **repeat n+1** termina.
–PEER REVIEW–

Esercizio 2 Una lista di interi è definita dalla seguente sintassi

$$L ::= (n, 0)|(n, L)$$

Ad esempio, la lista con gli interi 3 e 5 è rappresentata dal termine $(3, (5, 0))$. Si definisca un termine HOFL t (chiuso e tipabile) tale che l'applicazione $(t\ L)$ ad una lista L di 3 elementi ritorni l'ultimo elemento della lista.
è possibile dare un termine HOFL (chiuso e tipabile) che calcoli l'ultimo elemento di una generica lista di interi?

Svolgimento

$$(3,(5,0))$$

$$(t(n_1(n_2(n_3,0)))) = n_3 \qquad t = (\lambda l.\,\mathbf{fst}(\mathbf{snd}(\mathbf{snd}(l))))$$

Dobbiamo vedere che t sia ben tipato.

$$\begin{array}{l}
\text{lista} = int * (int * (int * int)) \\
\mathbf{snd}(l) = int * (int * int) \\
\mathbf{snd}(\mathbf{snd}(l) = int * int \\
\mathbf{fst}(\mathbf{snd}(\mathbf{snd}(l))) = int
\end{array}$$

Non è possibile per questione dei tipi generare l'ultimo elemento dei tipi. Una generica lista ha un tipo qualunque mentre i nostri termini hanno un tipo prefissato.
Una soluzione ipotetica è la seguente

$$\mathbf{rec}\,f.\ \underbrace{\lambda x}_{(\tau*int)\to\tau}\ .\ \underbrace{\mathbf{if}\ \mathbf{snd}(x)\ \mathbf{then}\ \underbrace{\mathbf{fst}(x)}_{\tau}\ \mathbf{else}(\ \underbrace{f}_{int\to\tau}\ \underbrace{\mathbf{snd}(\ \underbrace{x}_{\tau*int}\ )}_{int})}_{\tau}$$

Ma questa cosa non è tipabile. **if snd**(x) deve essere intero.

$$\begin{array}{ll}
x: & \tau * int \\
\mathbf{fst}(x): & \tau \\
\lambda x: & \tau * int \\
f: & int \to \tau \\
\mathbf{if}\ \mathbf{snd}(x)\ \mathbf{then}\ \mathbf{fst}(x)\ \mathbf{else}(f\ \mathbf{snd}(x)): & \tau \\
\lambda x.\ \mathbf{if}\ \mathbf{snd}(x)\ \mathbf{then}\ \mathbf{fst}(x)\ \mathbf{else}(f\ \mathbf{snd}(x)): & (\tau * int) \to \tau
\end{array}$$

rec f e λ devono avere lo stesso tipo, otteniamo quindi

$$\tau * int \to \tau = int \to \tau \quad \Longrightarrow \quad \tau * int = int$$

Non è possibile unificare, fallimento.

Esercizio 3 Dati due termini

$$t_1 \equiv \lambda x.\lambda y.x + 3$$

$$t_2 \equiv \lambda z.\,\mathbf{fst}(z) + 3$$

- Se ne calcolino i tipi
- Si verifiche se, data la forma canonica $c : \tau$, i due termini

  $$((t_1\ \ 1)c) \quad (t_2(1,c))$$

  si riducono alla stessa forma canonica.

Svolgimento

$$t_1 = \lambda\,\underbrace{\underbrace{x}_{int}\,.\lambda\,\underbrace{y}_{\tau}\,.\,\underbrace{\underbrace{x}_{int}+3}_{int}}_{int\to\tau\to int}$$

$$t_2 = \lambda \underbrace{z.\,\underbrace{\underbrace{\mathbf{fst}(\ \underbrace{z}_{int*\tau}}_{int}\ )+3}_{int}}_{(int*\tau)\to int}$$

Bisogna dimostrare

$$((t_1 \ \ 1)c) \to c' \qquad \wedge \qquad (t_2(1,c)) \to c'$$

$$\mathscr{C}[\![c]\!]^{n-1}\sigma'' \stackrel{?}{=} \sigma'$$

Prima riduzione: si parte dall'esterno.

$$(((\lambda x.\lambda y.x+3) \ \ 1) \ \ c) \to c'$$
$$\xLeftarrow{\text{inf. lazy}} ((\lambda x.\lambda y.x+3)1) \to \lambda x.t \quad t[^c/_x] \to c' \qquad (\lambda x \text{ a sinistra e a destra sono diversi})$$
$$\xLeftarrow{\text{inf. lazy}} (\lambda y.x+3)[^1/_x] \to \lambda x.t \quad t[^c/_x] \to c' \qquad (\text{Calcolo forma canonica perché 1 è canonico})$$
$$\Leftarrow (\underline{\lambda y}.1+3) \to \underline{\lambda x}.t \quad t[^c/_x] \to c' \equiv (1+3)[^c/_x] \to c' \qquad (\lambda x \text{ e } \lambda y \text{ sono unificabili})$$
$$\Leftarrow 1+3 \to c'$$
$$\xLeftarrow{c'=c''+c'''} 1 \to c'' \quad 3 \to c'''$$
$$\Leftarrow c''=1 \quad c'''=3 \quad \square$$

λx a sinistra e a destra sono diversi: il significato è "stiamo cercando un qualcosa che abbia forma di un astrazione" (vedi regola valutazione lazy).

Seconda riduzione

$$((\lambda z.fst(z)+3)(1,c)) \to c'$$
$$\Leftarrow (\lambda z.fst(z)+3) \to \lambda x.t \quad t[^{(1,c)}/_x] \to c'$$
$$\Leftarrow (fst(z)+3)[^{(1,c)}/_z] \to c'$$
$$\Leftarrow (fst(1,c)+3) \to c'$$
$$\xLeftarrow{c'=c''+c'''} fst(1,c) \to c'' \quad 3 \to c'''$$
$$\xLeftarrow{c'''=3} fst(1,c) \to c'' \quad 3 \to c''$$
$$\xLeftarrow{c'''=1} 1 \to c'' \quad 3 \to c'''$$

B.0.3. 2 Febbraio 2000

Esercizio 1 Si aggiunga a IMP il comando

$$\textbf{do}\ c\ \textbf{undoif}\ b$$

con il seguente significato informale: esegui il comando c, se dopo l'esecuzione di c l'espressione booleana b risulta soddisfatta allora ripristina lo stato precedente all'esecuzione di c.

1. Si definisca la semantica operazionale del nuovo comando
2. Si definisca la semantica denotazionale del nuovo comando
3. Si modifichi la prova di equivalenza fra semantica operazionale e denotazionale per tener conto del nuovo comando

Svolgimento

Punto a

$$\frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma\rangle \to \textbf{false}}{\langle \textbf{do}\ c\ \textbf{undo if}\ b,\sigma\rangle \to \sigma'} \qquad \frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma\rangle \to \textbf{true}}{\langle \textbf{do}\ c\ \textbf{undo if}\ b,\sigma\rangle \to \sigma}$$

Punto b

$$\mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma = \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to \underbrace{\sigma}_{\textbf{true}}, \underbrace{\mathscr{C}[\![c]\!]\sigma}_{\textbf{false}}$$

Punto c Dobbiamo dimostrare i due sensi come al solito

Dimostriamo da denotazionale a operazionale
Utilizziamo l'induzione strutturale. Dobbiamo dimostrare:

$$P(\textbf{do}\ c\ \textbf{undo if}\ b) = \mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma = \sigma' \overset{?}{\Longrightarrow} \langle \textbf{do}\ c\ \textbf{undo if}\ b, \sigma\rangle \to \sigma'$$

Anche qui possiamo sempre assumere le premesse, in questo caso sulle strutture già dimostrate. Abbiamo due casi possibili, **true** e **false**.

- $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$

  Per ipotesi induttiva sulle strutture sappiamo che

  $$P(b) = \mathscr{B}[\![b]\!]\sigma = \textbf{false} \Longrightarrow \langle b, \sigma\rangle \to \textbf{false}$$

  Inoltre, sempre per induzione strutturale.

  $$P(c) = \mathscr{C}[\![c]\!]\sigma = \sigma' \Longrightarrow \langle c, \sigma\rangle \to \sigma'$$

  Abbiamo ottenuto le premesse del nostro comando nella semantica operazionale di IMP, quindi OK.

- $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$
  Simmetrico al precedente.

Dimostriamo da operazionale a denotazionale
Utilizziamo induzione sulle regole. 2 Regole da dimostrare:

$$\frac{\langle c, \sigma\rangle \to \sigma' \quad \langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{do}\ c\ \textbf{undo if}\ b, \sigma\rangle \to \sigma'} \qquad \frac{\langle c, \sigma\rangle \to \sigma' \quad \langle b, \sigma\rangle \to \textbf{true}}{\langle \textbf{do}\ c\ \textbf{undo if}\ b, \sigma\rangle \to \sigma}$$

- **Prima regola**

  $$P(\langle \textbf{do}\ c\ \textbf{undo if}\ b, \sigma\rangle \to \sigma') \overset{\text{def}}{=} \mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma \overset{?}{=} \sigma'$$

  Supponiamo che P valga per le premesse. Sappiamo quindi, per ipotesi induttiva, che:

  $$\mathscr{C}[\![c]\!]\sigma = \sigma' \qquad \mathscr{B}[\![b]\!]\sigma = \textbf{false}$$

  Questo ci garantisce che il comando termina, possiamo togliere la stellina. Inoltre, siamo nel caso false, quindi:

  $$\mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma = \mathscr{B}[\![b]\!](\mathscr{C}[\![c]\!]\sigma) \to \underbrace{\mathscr{C}[\![c]\!]\sigma}_{\textbf{false}} = \sigma'$$

- **Seconda regola**

  $$P(\langle \textbf{do}\ c\ \textbf{undo if}\ b, \sigma\rangle \to \sigma) \overset{\text{def}}{=} \mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma \overset{?}{=} \sigma$$

  Premesse:

  $$\mathscr{C}[\![c]\!]\sigma = \sigma' \qquad \mathscr{B}[\![b]\!]\sigma = \textbf{true}$$

  Questo ci garantisce che il comando termina, possiamo togliere la stellina. Allora

  $$\mathscr{C}[\![\textbf{do}\ c\ \textbf{undo if}\ b]\!]\sigma = \mathscr{B}[\![b]\!](\mathscr{C}[\![c]\!]\sigma) \to \underbrace{\sigma}_{\textbf{true}} = \sigma$$

Esercizio 2 Si definiscano concretamente tre funzioni $f_i : D_i \to D_i$ con D_i cpo, per $i \in \{1, 2, 3\}$ tali che

- f_1 è continua, ha punti fissi, ma non ha minimo punto fisso;
- f_2 è continua, e non ha punti fissi;
- f_3 è monotona ma non continua.

TODO

B.0.4. 20 Gennaio 2005

Esercizio 1 Si dimostri che, per ogni $\sigma, \sigma' \in \Sigma$, se

$$\langle \mathbf{while}\, x \neq 0 \lor y \neq 0\, \mathbf{do}\, x := x - 1; y := y - 1, \sigma\rangle \to \sigma'$$

allora

$$\sigma(x) = \sigma(y) \land \sigma' = \sigma[^0/_x, ^0/_y]$$

Svolgimento La dimostrazione si esegue per induzione sulle regole.

$$P(\langle \mathbf{w}, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) = \sigma(y) \geq 0 \lor \sigma' = \sigma[^0/_x, ^0/_y]$$

- **Prima regola del while**

$$\frac{\langle x \neq 0 \lor y \neq 0, \sigma\rangle \to \mathbf{false}}{\langle \mathbf{w}, \sigma\rangle \to \sigma}$$

  Assumiamo la premmessa $\sigma(x) \neq 0 \lor \sigma(y) \neq 0 = \mathbf{false}$, cioè $\sigma(x) = 0$ e $\sigma(y) = 0$
  Dobbiamo dimostrare $P(\langle \mathbf{w}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \sigma(x) \stackrel{?}{=} \sigma(y) \geq 0$ e $\sigma \stackrel{?}{=} \sigma[^0/_x, ^0/_y]$. Ovvio

- **Seconda regola del while**

$$\frac{\langle x \neq 0 \lor y \neq 0, \sigma\rangle \to \mathbf{true} \quad \langle x := x - 1; y := y - 1, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle \mathbf{w}, \sigma\rangle \to \sigma'}$$

  Assumiamo

$$x \neq 0 \lor y \neq 0 \qquad \sigma'' = \sigma[^{\sigma(x)-1}/_x, ^{\sigma(y)-1}/_y]$$

$$P(\langle \mathbf{w}, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma''(x) = \sigma''(y) \geq 0 \land \sigma' = \sigma''[^0/_x, ^0/_y]$$

  sostituendo

$$\sigma(x) - 1 = \sigma(y) - 1 \geq 0 \qquad \sigma(x) = \sigma(y) \geq 1$$

$$\sigma' = \sigma[^{\sigma(x)-1}/_x, ^{\sigma(y)-1}/_y][^0/_x, ^0/_y] = \sigma[^0/_x, ^0/_y]$$

  dobbiamo dimostrare

$$P(\langle \mathbf{w}, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) = \sigma(y) \geq 0 \land \sigma' = [^0/_x, ^0/_y]$$

  ovvio.

Esercizio 2 Sia $(\omega \cup \{\infty\}, \geq)$ il cpo con bottom costituito dai naturali più infinito con l'ordinario ordinamento, però invertito. Inoltre sia $(\mathscr{P}_f(\omega \cup \{\infty\}), \subseteq)$ l'ordinamento parziale dai sottoinsiemi finiti di $\omega \cup \{\infty\}$ ordinati per inclusione.

1. Si dimostri con un controesempio che $(P_f(\omega \cup \{\infty\}), \subseteq)$ non è completo
2. Si dimostri che l'ordinamento parziale $(\mathscr{P}_f(\omega \cup \{\infty\}) \cup \{\omega \cup \{\infty\}\}, \subseteq)$ è completo
3. Si dimostri che la funzione

$$\min : (\mathscr{P}_f(\omega \cup \{\infty\}) \cup \{\omega \cup \{\infty\}\}, \subseteq) \to (\omega \cup \{\infty\}, \geq)$$

   che calcola in valore minimo di un insieme (con $\min(\varnothing) = \infty$) è monotona, ma si faccia vedere con un controesempio che non è continua

Svolgimento

Punto 1 Come sappiamo un ordinamento parziale è detto completo se ogni sua catena ha lub. Prendiamo come catena la seguente:

$$\{0\} \subseteq \{0,2\} \subseteq \{0,2,4\} \subseteq \dots$$

ossia

$$\{S_i\}_{i\in\omega} \qquad \text{con} \qquad S_i = \{n \leq 2i, n \text{ pari}\} \in P_f(\omega \cup \{\infty\})$$

Questa catena non ha maggioranti in $P_f(\omega \cup \{\infty\})$. Non troveremo mai un maggiorante. Il motivo per cui non troveremo mai un maggiorante è che i sottoinsiemi sono richiesti finiti, quindi troveremo sempre un elemento più grande un candidato ad essere maggiorante.

Punto 2 Data una qualunque catena finita $\{S_i\}_{i\in\omega}$ il suo *lub* è S_w con $S_w = S_{n+k}, k = 0, 1, \dots$. Data una qualunque catena infinita $\{S_i\}_{i\in\omega}$ questa ha un solo un maggiorante in $(\mathscr{P}_f(\omega \cup \{\infty\}) \cup \{\omega \cup \{\infty\}\}, \subseteq)$, ossia $\{\omega \cup \{\infty\}\}$

Punto 3 Monotonia

$$S_1 \subseteq S_2 \overset{?}{\Longrightarrow} \min(S_1) \geq \min(S_2)$$

Ovvio: $S_2 = S_1 \cup S$, $\min(S_2) = \min(\min(S_1), \min(S)) \leq \min(S_1)$

Continuità.
Data la catena $\{S_i\}_{i\in\omega}, S_i = \{n\}$, abbiamo che $\min(S_0) = \infty$ e $\min(S_i) = 1 \quad \forall i = 1, 2, \dots$ Quindi $\bigsqcup_{i\in\omega} \min(S_i) = 1$. Invece $\cup S_i = \omega$ (L'insieme infinito). Ma allora $\min(\bigsqcup S_i) = \min \omega = 0$.

Esercizio 3 Si verifichi se il seguente termine HOFL è tipabile, ed eventualmente se ne fornisca il tipo.

$$\textbf{rec}\, f.\lambda.\ \textbf{if snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ 1 + (f\ \textbf{snd}(x))$$

Svolgimento

$\mathbf{s}nd(x)$:	int
x :	$\tau * int$
$1 + (f\ \textbf{snd}(x))$:	int
$\textbf{if snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ 1 + (f\ \textbf{snd}(x))$:	int
$\lambda.\ \textbf{if snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ 1 + (f\ \textbf{snd}(x))$:	$\tau * int$

$$int \to int = \tau * int \to int$$
$$int = \tau * int$$

Assurdo!

B.0.5. Prova scritta del 6 aprile 2005

Esercizio 1 Si considerino i comandi:

$$w = \textbf{while}\ y < n\ \textbf{do}(x := x + 2y + 1; y := y + 1) \qquad \text{e} \qquad c = x := 0; y := 0; w$$

Si dimostri che, per ogni $\sigma, \sigma' \in \Sigma$ con $\langle w, \sigma\rangle \to \sigma'$,

$$\text{se } \sigma(x) = \sigma(y)^2, 0 \leq \sigma(y) \leq n$$

$$\text{allora } \sigma' = \sigma[^{n^2}/_x,^{n}/_y].$$

Si dimostri quindi che se $0 \leq n$ allora $\langle c, \sigma\rangle \to \sigma[^{n^2}/_x,^{n}/_y]$.

Svolgimento Dimostrazione per induzione sulle prove

$$P(\langle \mathbf{w}, \sigma \to \sigma') \stackrel{\text{def}}{=} \sigma(x) = \sigma(y)^2, 0 \le \sigma(y) \le n \Longleftarrow \sigma' = \sigma[^{n^2}/_x, ^n/_y]$$

Prima regola del while Usiamo la regola in cui la valutazione della guardia ha valore **false**. Dovrà essere quindi

$$\frac{\sigma(y) \ge n}{\langle \mathbf{w}, \sigma \rangle \to \sigma} \qquad P(\langle \mathbf{w}, \sigma \to \sigma') \stackrel{\text{def}}{=} \sigma(x) = \sigma(y)^2, 0 \le \sigma(y) \le n \stackrel{?}{\Longrightarrow} \sigma' = \sigma[^{n^2}/_x, ^n/_y]$$

ma la proprietà è ovvia in quanto

$$\sigma(y) \ge n \wedge \sigma(y) \le n \quad \Longrightarrow \quad \sigma(y) = n$$

□

Seconda regola del while Dovremo avere

$$\frac{\sigma(y) < n \quad \langle x := x + 2y + 1; y = y + 1, \sigma \rangle \to \sigma'' \quad \langle \mathbf{w}, \sigma'' \rangle \to \sigma'}{\langle \mathbf{w}, \sigma \rangle \to \sigma'}$$

assumiamo $\sigma(y) < n$

$$\sigma''(x) = \sigma''(y)^2, 0 \le \sigma''(y) \le n \Longrightarrow \sigma' = \sigma''[^{n^2}/_x, ^n/_y] \qquad \text{ipotesi induttiva}$$

dobbiamo dimostrare

$$\sigma(x) = \sigma(y)^2, 0 \le \sigma(y) \le n \stackrel{?}{\Longrightarrow} \sigma' = \sigma[^{n^2}/_x, ^n/_y]$$

assumiamo le premesse: $\sigma(x) = \sigma(y)^2, 0 \le \sigma(y) \le n$
da dimostrare la conseguenza: $\sigma' \stackrel{?}{=} \sigma[^{n^2}/_x, ^n/_y]$

$$\begin{array}{l} \sigma''(x) = \sigma(x) + 2\sigma(y) + 1 \qquad \sigma''(y) = \sigma(y) + 1 \\ \sigma''(y)^2 = \sigma(y)^2 + 2\sigma(y) + 1 = \sigma(x) + 2\sigma(y) + 1 = \sigma''(x) \\ 0 \le \sigma(y) \le n \quad \sigma(y) < n \quad \text{quindi} \quad 0 \le \sigma(y) < n \quad 0 < \sigma(y) + 1 \le n \quad 0 < \sigma''(y) \le n \end{array}$$

Abiamo quindi dimostrato le premesse dell'ipotesi induttiva $\sigma' = \sigma''[^{n^2}/_x, ^x/_y]$. Abbiamo quindi la conseguenza: $\sigma' = \sigma''[^{n^2}/_x, ^n/_y]$ cioè

$$\sigma' = \sigma[^{\sigma(x)+2\sigma(y)+1}/_x, ^{\sigma(y)+1}/_y][^{n^2}/_x, ^n/_y] = \sigma[^{n^2}/_x, ^n/_y]$$

□
Per la seconda parte, assumiamo $0 \le n$ e usiamo la regola del ";"

$$\frac{\langle x := 0; y := 0;, \sigma \rangle \to \sigma'' \qquad \langle \mathbf{w}, \sigma'' \rangle \to \sigma'}{\langle \mathbf{c}, \sigma \rangle \to \sigma'} \qquad \sigma'' = \sigma[^0/_x, ^0/_y]$$

dimostriamo che $\langle \mathbf{w}, \sigma[^0/_x, ^0/_y] \rangle \stackrel{?}{\Longrightarrow} \sigma[^{n^2}/_y, ^0/_x]$
Infatti vale senz'altro

$$\sigma[^0/_x, ^0/_y](x) = \sigma[^0/_x, ^0/_y](y)^2 \qquad 0 \le \sigma[^0/_x, ^0/_y](y) \le n$$

quindi, in base alla dimostrazione precedente, $\sigma' = \sigma[^{n^2}/_x, ^n/_y]$
avendo dimostrato la premessa, vale la conseguenza del ";", ossia $\langle c, \sigma \rangle \to \sigma[^{n^2}/_x, ^n/_y]$.

B.0.6. Prova scritta del 21 luglio 2005

Esercizio 1 Si consideri il comando:

$$w = \textbf{while}\, n \neq m\, \textbf{do if}\, n < m\, \textbf{then}\, m := m - n\, \textbf{else}\, n := n - m$$

Si dimostri che, per ogni $\sigma, \sigma' \in \Sigma$ con $\langle w, \sigma\rangle \to \sigma', \sigma(n), \sigma(m) > 0$ e per ogni $d > 0$:

$$d|\sigma(n), d|\sigma(m) \qquad \Longleftrightarrow \qquad \sigma'(n) = \sigma'(m) > 0, d|\sigma'(n)$$

dove $i|j$ significa che i è un divisore di j, cioè $\exists k. j = ki$.
Si dimostri infine che $\sigma'(n) = MCD(\sigma(n), \sigma(m))$ dove $k = MCD(i, j)$, con k, i, j naturali, è il massimo comun divisore di i e j, cioè $k|i$ e $k|j$ se e solo se $k|MCD(i, j)$.

Svolgimento Per induzione sulle regole

$$P(\langle \mathbf{w}, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall\sigma : \sigma(n), \sigma(m) > 0. \forall d > 0. d|\sigma(n), d|\sigma(m) \Leftrightarrow \sigma'(n) = \sigma'(m) > 0 \quad d|\sigma'(n)$$

Prima regola del while

$$\frac{\langle n \neq m, \sigma\rangle \to F}{\langle \mathbf{w}, \sigma\rangle \to \sigma} \qquad \sigma(n) = \sigma(m) \quad \text{per le premesse}$$

$$P(\langle \mathbf{w}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall\sigma : \sigma(n), \sigma(m) > 0. \forall d > 0. d|\sigma(n), d|\sigma(m) \Leftrightarrow \sigma(n) = \sigma(m) > 0 \quad d|\sigma(n)$$

ovvio per le premesse □

Seconda regola del while

$$\frac{\langle n \leq m, \sigma\rangle \to T \quad \langle \textbf{if}\, n < m\, \textbf{then}\, m := m - n\, \textbf{else}\, n := n - m, \sigma\rangle \to \sigma'' \quad \langle \mathbf{w}, \sigma''\rangle \to \sigma'}{\langle \mathbf{w}, \sigma\rangle \to \sigma'}$$

$\sigma(n) = \sigma(m)$

$$\sigma'' = \sigma(n) < \sigma(m) \to \sigma[{}^{\sigma(m)-\sigma(n)}/_m], \sigma[{}^{\sigma(n)-\sigma(m)}/_n]$$

$$P(\langle \mathbf{w}, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall\sigma : \sigma(n), \sigma(m) > 0 \stackrel{?}{\Longrightarrow} d|\sigma(n), d|\sigma(m) \Leftrightarrow \sigma'(n) = \sigma'(m) > 0 \quad d|\sigma'(n)$$

se riusciamo a dimostrare le premesse del $\langle \textbf{while}, \sigma''\rangle \to \sigma'$ siamo a posto

- **Caso** $\sigma(n) < \sigma(m)$

$$\sigma'' = \sigma[{}^{\sigma(m)-\sigma(n)}/_m] \qquad \sigma''(n) = \sigma(n) \quad \sigma''(m) = \sigma(m) - \sigma(n)$$

assumiamo le premesse della tesi $\sigma(n), \sigma(m) > 0$ dimostriamo la premessa dell'ipotesi

$\sigma''(n) > 0$ ovvio essendo $\sigma''(n) = \sigma(n)$
$\sigma''(m) > 0$ ovvio essendo $\sigma(n) < \sigma(m)$ e $\sigma''(m) = \sigma(m) = \sigma(n)$

Dall'ipotesi induttiva abbiamo quindi

$$d|\sigma''(n), d|\sigma''(m) \stackrel{?}{\Leftrightarrow} \sigma'(n) = \sigma'(m) > 0, d|\sigma'(n)$$

Pertanto basta dimostrare:

$$d|\sigma''(n), d|\sigma''(m) \stackrel{?}{\Leftrightarrow} d|\sigma(n), d|\sigma(m)$$

Cioè

$$\exists k_n.\sigma(n) = k_n d, \quad \exists k_m.\sigma(m) = \sigma(n) = k_m d$$

se e solo se?

$$\exists k_n.\sigma(n) = h_n d, \quad \exists k_m.\sigma(n) = h_m d$$

Infatti basta porre $h_n = k_n$ e $h_m = k_n + k_m$.

- **Caso** $\sigma(n) \geq \sigma(m)$
  La prova è analoga.

La dimostrazione è completa.
Possiamo concludere che prendendo $\sigma(n), \sigma(m)$, e d come naturali, abbiamo che tutti e soli i divisori comuni di $\sigma(n)$ e $\sigma(m)$ sono anche divisori di $\sigma'(n)$ che quindi è il MCD, essendo un naturale.

B.0.7. Prova scritta del 3 Aprile 2007

Esercizio 1 Si condiseri il linguaggio IMP senza comandi di assegnamento Si dimostri per induzione strutturale sui comandi c che per ogni c, σ:

$$\langle c, \sigma\rangle \to \sigma' \quad \text{oppure} \quad \langle c, \sigma\rangle \not\to$$

Dimostrare esplicitamente il caso di non terminazione utilizzando la regola di inferenza vista a lezione

Svolgimento

$$P(c) \overset{\text{def}}{=} \forall \sigma. \langle c, \sigma\rangle \to \sigma \vee \langle c, \sigma\rangle \not\to$$

Come al solito andiamo per casi

- $P(\textbf{skip}) \overset{\text{def}}{=} \forall \sigma. \underbrace{\langle \textbf{skip}, \sigma\rangle \to \sigma}_{?} \vee \underbrace{\langle \textbf{skip}, \sigma\rangle \not\to}_{?}$
  Ovvio essendo $\langle \textbf{skip}, \sigma\rangle \to \sigma$

- $P(c_0; c_1) \overset{\text{def}}{=} \forall \sigma. \underbrace{\langle c_0; c_1, \sigma\rangle \to \sigma}_{?} \vee \underbrace{\langle c_0; c_1, \sigma\rangle \not\to}_{?}$
  per ipotesi induttiva abbiamo

  $$P(c_i) \overset{\text{def}}{=} \forall \sigma. \langle c_i, \sigma\rangle \to \sigma \vee \langle c_i, \sigma\rangle \not\to$$

  dobbiamo andare per casi

  - **Caso** $\langle c_i, \sigma\rangle \not\to$
    Allora anche $\langle c_0; c_1, \sigma\rangle \not\to$ dato che la regola richiede $\langle c_0, \sigma \to\rangle \to \sigma''$ e $\langle c_1, \sigma''\rangle \to \sigma'$, con $\sigma = \sigma'' = \sigma'$
  - **Caso** $\langle c_0, \sigma\rangle \to \sigma$ e $\langle c_1, \sigma\rangle \to \sigma$
    Allora $\langle c_0; c_1, \sigma\rangle \to \sigma$ per la regola.

- $P(\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1) \overset{\text{def}}{=} \forall \sigma. \underbrace{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \to \sigma}_{?} \vee \underbrace{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \not\to}_{?}$
  se $b = \textbf{true}$ allora la tesi è dimostrata dalla regola

  $$\frac{\langle v, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \to \sigma}$$

  essendo

  $$\langle c_0, \sigma\rangle \to \sigma \iff \langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \to \sigma$$

  $$\langle c_0, \sigma\rangle \not\to \iff \langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \not\to$$

  Sicuramente OK per $b = \textbf{false}$

- $P(\textbf{while}\ b\ \textbf{do}\ c) \overset{\text{def}}{=} \forall \sigma. \underbrace{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \to \sigma}_{?} \vee \underbrace{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \not\to}_{?}$

  - **Caso** $\langle b, \sigma\rangle \to \textbf{false}$

    $$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \to \sigma} \qquad \text{ovvio}$$

☠ **Caso** $\langle b, \sigma \rangle \to \textbf{true}$
prendiamo in considerazione la regola di non terminazione

$$\frac{\forall \sigma \in S.\langle c, \sigma \rangle \to \sigma' \quad \sigma' \in S \quad \forall \sigma \in S.\langle b, \sigma \rangle \to \textbf{true}}{\langle w, \sigma \rangle \not\to}$$

prendiamo $S = \{\sigma\}$

- ⋆ $\langle c, \sigma \rangle \to \sigma' \overset{?}{\Rightarrow} \sigma = \sigma'$ per l'ipotesi induttiva ci sono due possibilità
  - ⋄ $\langle c, \sigma \rangle \not\to$: vero essendo la premessa falsa.
  - ⋄ $\langle c, \sigma \rangle \to \sigma'$: vero essendo $\sigma = \sigma'$.
- ⋆ $\langle b, \sigma \rangle \to \textbf{true}$: vero nel caso in questione.

Esercizio 2 Si consideri l'isieme $\mathscr{D}$ contenente tutti gli insiemi di numeri naturali composti solamente da numeri dispari. Si dimostri che $(\mathscr{D}, \subseteq)$ è un ordinamento parziale completo con bottom. Si dica inoltre quali insiemi $S \in \mathscr{D}$ di naturali possono essere il limite di quache catena infinita.

Svolgimento $(\mathscr{D}, \subseteq)$ è po dato che $\mathscr{D}$ è un sottoinsieme di $\mathscr{P}(\omega)$ che è un po e l'ordinamento è lo stesso.
Ha bottom: $\emptyset \subseteq S, \forall S \in \mathscr{D}$
è completo: Sia S_i una catena abbiamo che $\bigcup_{i \in \omega} S_i \overset{\text{def}}{=} \{n | \exists k.n \in S_k\}$ ma se S_k contiene solo numeri dispari, anche $\bigcup_{i \in \omega} S_i$ contiene solo numeri dispari.
I limiti delle catene infinite sono tutti e soli gli insieme infiniti di numeri dispari.
Sia $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$ una catena con $\{S_i\}$ infinito, allora non può essere $|\bigcup_{i \in \omega} S_i| = N$ perché i sottoinsiemi di un insieme finito sono in numero finito, mentre deve essere $\forall i.S_i \subseteq \bigcup_{i \in \omega} S_i$.
Dato S infinito la catena

$$S \cap \emptyset \subseteq S \cap \{0\} \subseteq S \cap \{0, 1\} \subseteq S \cap \{0, 1, 2\} \subseteq \ldots$$

è una catena infinita con $\bigcup_{i \in \omega} S_i = S$.

Esercizio 3 Dato il comando del linguaggio IMP

$$\textbf{while}\ x \neq 0\ \textbf{do}(x := x - 1; y := y + 1)$$

si dimostri usando l'induzione computazionale di Scott che, dati comunque gli stati σ e σ',

$$\mathscr{C}[\![w]\!]\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^{\sigma x + \sigma y}/_x, ^0/_x]$$

TODO: è identico ad un esempio dell'induzione computazionale visto a lezione, quindi si può anche non copiare.

B.0.8. Prova scritta del 31/05/06

Esercizio 1

$$\mathscr{C}[\![\textbf{do}\ c_0\ \textbf{if diverge}\ c_1]\!]\sigma = (\mathscr{C}[\![c_0]\!]\sigma \in \Sigma) \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma$$

TODO manca il tipaggio dell'espressione.

Il dominio $T_\perp$ è ottenuto *liftando* $\{true, false\}$ e il condizionale esteso è definito come estensione naturale sul primo argomento

- if true then x else y = x
- if false then x else y = y
- if $\perp_{T_\perp}$ then x else y = $\perp_{T_\perp}$

La funzione $\lambda x.x \in \Sigma$ è anch'essa un'estensione naturale

$$
\begin{aligned}
x \in \Sigma = &\text{case x} \\
& T : true \\
& \bot_{\Sigma_\bot} : \bot_{T_\bot}
\end{aligned}
$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'}{\langle \mathbf{do}\, c_0\, \mathbf{if\ diverge}\, c_1, \sigma\rangle \to \sigma'}$$

B.0.9. Prova scritta del 30/04/04

Esercizio 1 I numeri di Fibonacci $F(n), n \in \omega$ sono calcolati dalla seguente funzione ricorsiva:

$$F(0) = 0, \quad F(1) = 1, \quad F(n+2) = F(n) + F(n+1)$$

Questa definizione può essere vista come una definizione ben fondata? E per quale relazione ben fondata?
Si consideri ora il comando:

$$w = \mathbf{while}\, z < n\, \mathbf{do}\, v := y; y := x + y; x := v : z := z + 1.$$

Si dimostri che, per ogni $\sigma, \sigma' \in \Sigma$ con $\langle w, \sigma\rangle \to \sigma'$

$$\text{se} \quad \sigma(x) = F(\sigma(z)), \sigma(y) = F(\sigma(z) + 1), 0 \leq \sigma(z) \leq (n)$$

$$\text{allora} \quad \sigma'(x) = F(n), \sigma'(y) = F(n+1).$$

Si osservi quindi che $\langle x := 0; y := 1; z := 0; w, \sigma\rangle \to \sigma'$ con $n \geq 0$ implica $\sigma'(x) = F(n)$

Svolgimento La definizione è per ricorsione matematica nella versione completa, cioè la relazione ben fondata

$$n \leq m \qquad n, m \in \omega$$

Quindi $F(n)$ può essere definita in termini di tutte le F(k) con $k < n$. Nel caso specifico bastano $F(n-1)$ e $F(n-2), n \geq 2$.
La proprietà viene dimostrata per induzione sulle regole del while:

$$P(\langle w, \sigma \to \sigma') \stackrel{\text{def}}{=} \sigma(x) = F(\sigma(z)) \quad \sigma(y) = F(\sigma(z)+1) \quad 0 \leq \sigma(z) \Longrightarrow \sigma'(x) = F(n) \quad \sigma'(y) = F(n+1)$$

Prima regola del while Per la regola

$$\frac{\langle z < n, \sigma\rangle \to \mathbf{false}}{\langle w, \sigma\rangle \to \sigma'} \qquad \text{cioè con } \sigma(z) \geq n$$

va dimostrato

$$P(\langle w, \sigma \to \sigma) \stackrel{\text{def}}{=} \sigma(x) = F(\sigma(z)) \quad \sigma(y) = F(\sigma(z)+1) \quad 0 \leq \sigma(z) \Longrightarrow \sigma(x) = F(n) \quad \sigma(y) = F(n+1)$$

ma questo è ovvio, essendo $\sigma(z) = n$ (da $\sigma(z) \geq n$ e $\sigma(z) \leq n$).

Seconda regola del while

$$\frac{\langle z < n, \sigma\rangle \to \textbf{true} \quad \langle v := y; y := x + y; x := v : z := z + 1, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'}$$

assumendo $\sigma(z) < n$ per il test

$$P(\langle w, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma''(x) = F(\sigma''(z)) \quad \sigma''(y) = F(\sigma''(z){+}1) \quad 0 \le \sigma''(z) \Longrightarrow \sigma'(x) = F(n) \quad \sigma'(y) = F(n{+}1)$$

cioè, considerando la definizione di σ'' CHECK

$$\sigma'(x) = \sigma(y) = F(\sigma(z) + 1) \quad \sigma(y) = \sigma(x) + \sigma(y) = F(\sigma(z) + 2) \Longrightarrow \sigma'(x) = F(n) \quad \sigma'(y) = F(n + 1)$$

Possiamo quindi assumere le premesse della tesi

$$\sigma(x) = F(\sigma(z)) \quad \sigma(y) = F(\sigma(z) + 1) \quad 0 \le \sigma(z) \le n$$

con questa possiamo dimostrare le premesse dell'ipotesi

$$\begin{cases} \sigma(y) = F(\sigma(z) + 1) & \text{diretto} \\ \sigma(x) + \sigma(y) = F(\sigma(z) + 2) & \text{da } F(\sigma(z) + 2) = F(\sigma(z)) + F(\sigma(z) + 1) \\ 1 \le \sigma(z) \le n + 1 & \text{da } 0 \le \sigma(z) \text{ e } \sigma(z) < n \text{ per il test} \end{cases}$$

questo ci permette di assumere le conseguenze dell'ipotesi

$$\sigma'(x) = F(n) \quad \sigma'(y) = F(n + 1)$$

che è direttamente la conseguenza della tesi.

Infine abbiamo, con $n \ge 0$

$$\langle x := 0; y := 1; z := 0; w, \sigma\rangle \to \sigma' \longleftarrow \langle w, \sigma[^0/_x, ^1/_y, ^0/_z]\rangle \to \sigma'$$

Ma $\hat{\sigma} = \sigma[^0/_x, ^1/_y, ^0/_z]$ soddisfa la premessa.

Esercizio 2 Si consideri l'insieme PI delle funzioni parziali iniettive da ω a ω, con l'ordinamento $\sqsubseteq$ visto a lezione (inclusione degli insiemi di coppie cioè $f \sqsubseteq g$ se e solo se $Gr(f) \subseteq Gr(g)$, con $Gr(f) = \{\langle x, y\rangle | f(x) = y\}$). Se identifichiamo una funzione f con il suo grafo $Gr(f)$, abbiamo che f parziale iniettiva significa che $\langle x, y\rangle, \langle x, y'\rangle \in f \Rightarrow y = y'$ e $\langle x, y\rangle, \langle x', y\rangle \in f \Rightarrow x = x'$ iniettiva.

Si dimostri quindi che $(PI, \sqsubseteq)$ è un ordinamento parziale completo.

Si dimostri infine che la funzione $F : PI \to PI$ con $F(f) = \{\langle 2x, y\rangle | \langle x, y\rangle \in f\}$ è monotona continua. (Cenno: si condisderi F come calcolata dall'opratore $\hat{R}$ delle conseguenze immediate con R costituito dalla sola regola $\langle x, y\rangle / \langle 2x, y\rangle$)

Svolgimento Come al solito, per dimostrare che stiamo lavorando su un ordinamento parziale vanno dimostrate le seguenti prorietà: riflessività, antisimmetria, transività.

- **Proprietà riflessiva**: $f \sqsubseteq f$, ovvio essendo $f \subseteq f$
- **Proprietà antisimmetrica**: $f \sqsubseteq g, g \sqsubseteq f$, ovvio essendo $f \subseteq g$ e $g \subseteq f \Longrightarrow g = f$
- **Proprietà transitiva**: $f \sqsubseteq g, g \sqsubseteq h \stackrel{?}{\Longrightarrow}$, ovvio essendo $f \subseteq g$ e $g \subseteq g \Longrightarrow f \subseteq g$

Completezza Data una catena

$$f_0 \subseteq f_1 \subseteq \ldots f_i \subseteq \ldots$$

con $\langle x, y\rangle, \langle x, y'\rangle \in f_i \Longrightarrow y = y'$ e con $\langle x, y\rangle, \langle x', y\rangle \in f_i \Longrightarrow y = y'$
Bisogna dimostrare che anche il lub è parziale iniettivo

$$\langle x, y\rangle, \langle x, y'\rangle \in \bigcup_{i\in\omega} f_i \Longrightarrow y = y'$$

e

$$\langle x, y\rangle, \langle x', y\rangle \in \bigcup_{i\in\omega} f_i \Longrightarrow x = x'$$

Ma ciò è vero. Infatti se $\langle x, y\rangle, \langle x, y'\rangle \in \bigcup_{i\in\omega}$ allora esiste un k con $\langle x, y\rangle, \langle x, y'\rangle \in f_k$. Quindi $y = y'$ essendo f_k parziale. Vale banalmente il caso simmetrico.

Monotonia della funzione La funzione è continua in quanto il numero delle premesse è finito (vedi teorema di $\hat{R}$).

B.0.10. Prova scritta del 23/07/07

Esercizio 1 Un *contesto* C del linguaggio IMP è definito dalla seguente sintassi:

$$C ::= _;C;c|c;C|\,\mathbf{if}\ v\ \mathbf{then}\ C\ \mathbf{else}\ c|\,\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ C$$

Sia assuma $\mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$ e si dimostri che $\mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$ e si dimostri che $\mathscr{C}[\![C[c_1/_]]\!] = \mathscr{C}[\![C[c_2/_]]\!]$ per induzione strutturale su C. Si esamino soli i casi relativi a $_$, $C;c$ e **while** b **do** c. Si concluda quindi che la relazione $\equiv$ con $c_1 \equiv c_2 \Leftrightarrow \mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$ di equivalenza denotazionale è una congruenza rispetto alle operazioni dei comandi di IMP.

Svolgimento Assumiamo $\mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$. Dimostriamo

$$P(C) \stackrel{\text{def}}{=} \mathscr{C}[\![C[^{c_1}/_]]\!] = \mathscr{C}[\![C[^{c_2}/_]]\!]$$

- $P(_) \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!] = \mathscr{C}[\![c_2]\!]$ ovvio dalle premesse
- $P(C;c) \stackrel{\text{def}}{=} \mathscr{C}[\![C[^{c_1}/_];c]\!] = \mathscr{C}[\![C[^{c_2}/_];c]\!]$
  Per ipotesi induttiva $\mathscr{C}[\![C[^{c_1}/_]]\!] = \mathscr{C}[\![C[^{c_2}/_]]\!]$.
  Da dimostrare $\lambda\sigma.\mathscr{C}[\![c]\!]^*(\mathscr{C}[\![C[^{c_1}/_]]\!]\sigma) \stackrel{?}{=} \lambda\sigma.\mathscr{C}[\![c]\!]^*(\mathscr{C}[\![C[^{c_2}/_]]\!]\sigma)$
  Ma è ovvio per l'ipotesi induttiva.
- $P(\mathbf{while}\ b\ \mathbf{do}\ C) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{while}\ b\ \mathbf{do}\ C[^{c_1}/_]]\!] = \mathscr{C}[\![\mathbf{while}\ b\ \mathbf{do}\ C[^{c_2}/_]]\!]$
  Per ipotesi induttiva $\mathscr{C}[\![C[^{c_1}/_]]\!] = \mathscr{C}[\![C[^{c_2}/_]]\!]$.
  Da dimostrare che $fix\Gamma_1 \stackrel{?}{=} fix\Gamma_2$ con

  $$\Gamma_1\varphi\sigma = \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![C[^{c_1}/_]]\!]\sigma), \sigma$$

  $$\Gamma_2\varphi\sigma = \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![C[^{c_2}/_]]\!]\sigma), \sigma$$

  Ma anche qui la proprietà è ovvia per le premesse.

La relazione $\equiv$ è una congruenza se per tutte le operazioni f vale $c_1 \equiv c_2 \Longrightarrow f(c_1) \equiv f(c_2)$. I contesti $_;c$ e **if** b **then** _ **else** c e **if** b **then** c **else** _ e **while** b **do** _ rappresentano tutte le operazioni.

Esercizio 2 Data una segnatura Σ, si condiseri la relazione $\sqsubseteq\ = <^*$ tra termini $t \in T_\Sigma$ in Σ che è la chiusura riflessiva e transitiva della relazione $<$ termine - sottotermine definita come $t_i < f(t_1, \ldots, t_n), 1 \leq i \leq n$. Si dimostri che $\sqsubseteq$ è un ordinamento parziale. Si faccia vedere quindi, al variare di Σ, quando tale ordinamento è completo e/o con bottom.

Svolgimento Essendo $\sqsubseteq\ = \ <^*$ la proprietà riflessiva e simmetrica valgono per costruzione. La proprietà antissimetrica vale in quanto $<$ è una relazione aciclica (addirittura ben fondata) e l'operazione di chiusura non introduce cicli.

- **Caso** $c \in \Sigma_0$ **e** $f \in \Sigma_n$
  In tal caso T_Σ è un insieme infinito. La catena

  $$t_0 = c \quad \sqsubseteq \quad t_1 = f(t_0, \dots, t_0) \quad \sqsubseteq \quad t_2 = f(t_1, \dots, t_1) \quad \sqsubseteq \dots$$

  non ha nessun maggiorante in T_Σ.

- **Caso** $\Sigma_0 = \emptyset$
  In tal caso T_Σ è vuoto. Quindi completo e senza bottom.

- **Caso** $\Sigma_n = \emptyset, n \neq 0$
  Allora $\sqsubseteq$ è un ordinamento discreto ($t_1 \sqsubseteq t_2 \Longrightarrow t_1 = t_2$) quindi con catene finite (di un solo elemento), quindi completo.

- **Caso** $\Sigma_0 = \{c\}$ singleton
  L'ordinamento $(T_\Sigma, \sqsubseteq)$ ha c come bottom

- **Caso** Σ_0 non singleton
  $(T_\Sigma, \sqsubseteq)$ non ha minimo: vuoto se $\Sigma_0 = 0$ e più di un elemento minimale se Σ_0 ha due o più elementi.

Si ricordi che T_Σ è generato dalla seguente regola di inferenza:

$$\frac{t_i \in T_\Sigma \quad t = 1, \dots, n}{f(t_1, \dots, t_n) \in T_\Sigma}$$

B.0.11. Prova scritta del 9 aprile 2010

Esercizio 1 Si estenda IMP con : (i) la nuova categoria sintatattica S (con variabili s) delle espressioni *con effetti laterali*, brevemente *elat*, avente $\langle s, \sigma \rangle \to (\sigma', n)$ come formule ben formate per la semantica operazionale e dominio semantico $\Sigma \to (\Sigma \times N)_\perp$; e (ii) le nuove produzioni

$$s ::= c; a \qquad c ::= \ x := s$$

con la seguente semantica infromale : per la prima produzione, quella di ritornare come *stato modificato* della elat s lo stato prodotto dal comando c e come *risultato* di s la valutazione dell'espressione aritmetica a in tale stato; per la seconda produzione quella di vaulutare prima l'elat s e poi effettuare l'assegnamento utilizzando come memoria lo stato modificato da s, e come valore da assegnare ad x il risultato di s.

La semantica denotazionale è la seguente:

$$\mathscr{S}[\![c; a]\!]\sigma = (\lambda \sigma'.(\mathscr{A}\,[\![a]\!]\,\sigma;))^*(\mathscr{C}\,[\![c]\!]\,\sigma)$$

$$\mathscr{C}\,[\![x := s]\!]\,\sigma = \begin{cases} \textbf{case}\ \mathscr{S}[\![s]\!]\sigma\ \textbf{of} : & \\ \qquad \perp_{\Sigma \times N_\perp} : & \perp_{\Sigma_\perp} \\ \qquad (\sigma', n) : & \sigma'[^n/_x] \end{cases}$$

Si fornisca la semantica operazionale dei nuovi costrutti e si valuti il comando $x := (x := 3; 5)$ sia secondo la semantica operazionale sia secondo quella denotazionale. Si dimostri quindi l'equivalenza operazionale-denotazionale dei due nuovi costrutti e si faccia vedere (utilizzando a piacere la semantica operazionale oppure quella denotazionale) che $x := (c; a)$ è equivalente a $c; x := a$.

Svolgimento Semantica operazionale

$$\frac{\langle c, \sigma \rangle \to \sigma' \quad \langle a, \sigma' \rangle \to n}{\langle c; a, \sigma \rangle \to (\sigma', n)} \qquad \frac{\langle s, \sigma \rangle \to (\sigma', n)}{\langle x := s, \sigma \rangle \to \sigma'[^n/_x]}$$

Valutazione valutiamo $x := (x := 3; 5)$
Semantica operazionale

$$\begin{aligned}&\langle x := (x := 3;5), \sigma\rangle \to \sigma' \\ &\xleftarrow{\sigma=\sigma''[^n/_x]} \langle x := 3; 5, \sigma\rangle \to (\sigma'', n) \\ &\Longleftarrow \langle x := 3, \sigma\rangle \to \sigma'' \quad \langle 5, \sigma''\rangle \to n \xleftarrow{\sigma;;=\sigma[^3/_x} \square\end{aligned}$$

$\sigma' = \sigma''[^n/_x] = \sigma[^3/_x][^5/_x] = \sigma[^5/_x]$

Semantica denotazionale

$$\mathscr{C}\,[\![x := (x := 3; 5)]\!]\,\sigma = \begin{cases} \textbf{case} & (\sigma'.(\sigma', \mathscr{A}\,[\![5]\!]\,\sigma')^* \mathscr{C}\,[\![x := 3]\!]\, \textbf{of} : \\ & \bot_{\Sigma\times N_\bot} : \bot_{\Sigma_\bot} \\ & (\sigma', n) : \sigma'[^n/_x] \end{cases}$$

$$\begin{aligned}&= \textbf{let}\ (\sigma', n) = (\lambda\sigma'(\sigma', 5))\sigma[^3/_x]\ \textbf{in}\ \sigma''[^n/_x] \\ &= \textbf{let}\ (\sigma'', n) = (\sigma[^3/_x], 5)\ \textbf{in}\ \sigma''[^n/_x] \\ &= \sigma[^3/_x][^5/_x] = \sigma[^5/_x]\end{aligned}$$

Equivalenza semantiche Vediamo i due passi
Operazionale → denotazionale

$$\frac{\langle c, \sigma\rangle \to \sigma' \quad \langle a, \sigma'\rangle \to n}{\langle c; a, \sigma\rangle \to (\sigma', n)}$$

$\mathscr{S}[\![c; a, \sigma]\!]\sigma \stackrel{?}{=} (\sigma', n)$
$(\lambda\sigma''.(\sigma, \mathscr{A}\,[\![a]\!]\,\sigma''))^*(\mathscr{C}\,[\![c]\!]\,\sigma) \stackrel{?}{=} (\sigma', n)$
Si sa per ipotesi induttiva che $\mathscr{C}\,[\![c]\!]\,\sigma) \stackrel{?}{=} (\sigma', n)$. Quindi
$\lambda\sigma''.(\sigma''(, \mathscr{A}\,[\![a]\!]\,, \sigma'')\sigma' \stackrel{?}{=} (\sigma', n)$
$(\sigma', \mathscr{A}\,[\![a]\!]\,\sigma') \stackrel{?}{=} \sigma', n$
Vero, essendo $\mathscr{A}\,[\![a]\!]\,\sigma' = n$ per l'ipotesi induttiva.

$$\frac{\langle s, \sigma\rangle \to (\sigma', n)}{\langle x := s, \sigma\rangle \to \sigma'[^n/_x]}\ \mathscr{C}\,[\![x := s]\!]\,\sigma \stackrel{?}{=} \sigma'[^n/_x]$$

$$\mathscr{C}\,[\![x := s]\!]\,\sigma = \begin{cases} \textbf{case}\ \mathscr{S}[\![s]\!]\sigma\ \textbf{of} : \\ \qquad \bot_{\Sigma\times N_\bot} : \quad \bot_{\Sigma_\bot} & \stackrel{?}{=} \quad \sigma'[^n/_x] \\ \qquad (\sigma', n) : \quad \sigma'[^n/_x] \end{cases}$$

Ma $\mathscr{S}[\![s]\!]\sigma = (\sigma', n)$ per l'ipotesi induttiva, quindi $(\sigma', n) = (\sigma'', m)$, cioè $\sigma' = \sigma''$ e $n = m \quad \sigma''[^m/_x] \stackrel{?}{=} \sigma'[^n/_x]$

Denotazionale → Operazionale

$P(c; a) \stackrel{\text{def}}{=} \mathscr{S}[\![c; a]\!]\sigma = \sigma' \Longrightarrow \langle c; a, \sigma\rangle \to \sigma'$
$\mathscr{S}[\![c; a]\!]\sigma = (\sigma', n) \quad \langle c; a, \sigma\rangle \stackrel{?}{\Rightarrow} (\sigma', n)$
$(\lambda\sigma''.(\sigma'', \mathscr{A}\,[\![a]\!]\,\sigma''))^* \underbrace{(\mathscr{C}\,[\![c]\!]\,\sigma)}_{\sigma'''} = (\sigma', n) \quad \langle c, \sigma\rangle \to \sigma'''$
$(\lambda\sigma''.(\sigma''.\mathscr{A}\,[\![a]\!]\,\sigma''))\sigma''' = (\sigma', n)$
$(\sigma''.(\sigma''.\mathscr{A}\,[\![a]\!]\,\sigma))\sigma''' = (\sigma', n)$
$(\sigma''', \mathscr{A}\,[\![a]\!]\,\sigma''') = (\sigma', n) \quad \sigma''' = \sigma' \quad \mathscr{A}\,[\![a]\!]\,\sigma' = n$
Quindi $\langle c, \sigma\rangle \to \sigma' \quad \langle a, \sigma'\rangle \to n$

da cui $\langle c; a, \sigma\rangle \to \sigma'$ per la regola operazionale
$P(x := s) \stackrel{\text{def}}{=} \mathscr{C}[\![x := s]\!]\,\sigma = \sigma' \implies \langle x := s, \sigma\rangle \to \sigma'$
$\mathscr{C}[\![x := s]\!]\,\sigma = \sigma' \quad \langle x := s, \sigma\rangle \stackrel{?}{\to} \sigma'$
case $\mathscr{S}[\![s]\!]\sigma$ **of** :

$$\begin{array}{ll} \bot_{\Sigma\times N_\bot} : & \bot_{\Sigma_\bot} \\ (\sigma', n) : & \sigma'[^n/_x] = \sigma' \end{array}$$

let $\mathscr{S}[\![s]\!]\sigma = (\sigma'', n) \quad \langle x := s, \sigma\rangle \stackrel{?}{\to} \sigma''[^n/_x]$ Ovvio per la regola operazionale.

Equivalenza comandi $x := (c; a) \equiv c, x := a$
Operazionale

$$\begin{array}{l} \langle x := (c; a), \sigma\rangle \to \sigma' \\ \xleftarrow{\sigma'=\sigma''[^n/_x]} \langle c; a, \sigma\rangle \to (\sigma'', n) \\ \leftarrow \langle c, \sigma\rangle \to \sigma'' \quad \langle a, \sigma''\rangle \to n \end{array}$$

$$\begin{array}{l} \langle c; x := a, \sigma\rangle \to \sigma' \\ \leftarrow \langle c, \sigma\rangle \to \sigma'' \quad \langle x : a, \sigma''\rangle \to \sigma' \\ \xleftarrow{\sigma'=\sigma''[^m/_x]} \langle c, \sigma\rangle \to \sigma'' \quad \langle a, \sigma''\rangle \to \end{array}$$

Stessi sottogoal
Denotazionale

$$\mathscr{C}[\![x := (c; a)]\!]\,\sigma = \begin{cases} \textbf{case}(\lambda\sigma'.(\sigma', \mathscr{A}[\![a]\!]\,\sigma'))^*(\mathscr{C}[\![c]\!]\,\sigma)\ \textbf{of} : \\ \qquad \bot_{\Sigma\times N_\bot} : \quad \bot_{\Sigma_\bot} \\ \qquad (\sigma', n) : \quad \sigma'[^n/_x] = \sigma' \end{cases}$$

$\mathscr{C}[\![c; x := a]\!]\,\sigma = (\lambda\sigma'.\sigma'[^{\mathscr{A}[\![a]\!]\sigma'}/_x])^*\mathscr{C}[\![c]\!]\,\sigma$
Sono uguali: se $\mathscr{C}[\![c]\!]\,\sigma = \bot$ entrambi sono $\bot_{\Sigma_\bot}$ altrimenti altrimenti diventano $(\mathscr{C}[\![c]\!]\,\sigma)[^{\mathscr{A}[\![a]\!](\mathscr{C}[\![c]\!]\sigma)}/_c]$
□

Esercizio 2 Si consideri la grammatica libera (più precisamente lineare destra) $S ::= \alpha S|\beta$, dove l'alfabeto dei simboli terminali è $V = \{a, b\}$ e $\sigma, \beta \in V^*$. Si scriva il sistema di regola di inferenza R corrispondente a tale grammatica e si dimostri (in entrambi i sensi) che l'insieme dei teoremi I_R ne è il minimo punto fisso. Si dica infine se e quali altri punti fissi esistono per particolari valori di α e β.

Svolgimento

$$\frac{x \in S'}{\alpha x \in S} \qquad \beta \in S$$

Le formule ben formate sono $\{x \in S | x \in \{a, b\}^*\}$

$$P(\gamma \in S) \stackrel{\text{def}}{=} (\gamma \in S) \in \{\alpha^n\beta \in S | n \in \omega) \stackrel{\text{def}}{=} \exists n.\gamma = \alpha^n\beta$$

$P(\beta \in S)$ ovvio per $n = 0$
$P(\alpha x \in S) \stackrel{\text{def}}{=} \exists n.\alpha x \stackrel{?}{=} \alpha^n\beta$
Sappiamo però $P(x \in S) \stackrel{\text{def}}{=} \exists m.x = \alpha^m\beta$
quindi $\exists n.\alpha x = \alpha^n\beta$ vale con $n = m + 1$ □

Viceversa per induzione matematica

$\forall n.(\alpha^n\beta \in S) \in I_R$

$$\begin{array}{lll} P(O): & \beta \in S & \in I_R \\ P(n): & \alpha^n \in S & \in I_R \quad \text{ipotesi induttiva} \\ P(n+1): & \alpha\alpha^n \in S & \in I_R \quad \text{ovvio data la regola d'inferenza} \end{array}$$

$\hat{R}(X) = \{\beta \in S\} \cup \{\alpha x | x \in X\}$
$\hat{R}(\{\alpha^n\beta \in S | n \in \omega\}) = \{\beta \in S\} \cup \{\alpha^{n+1}\beta \in S | n \in \omega\} = \{\alpha^n\beta \in S | n \in \omega\}\}$ □

Se $\beta = \lambda$ stringa vuota, $X = \hat{R}(X)$ è soddisfatto da ogni $X \supseteq \{\beta \in S\}$.

Esercizio 3 I *coefficienti binomiali* $\binom{n}{k}$ con $n, k \in \omega$ e $0 \leq k \leq n$, sono definiti da

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Si dimostri che tale definizione è data per ricorsione ben fondata, fornendo la relazione ben fondata e la corrispondente $F(b, h)$. Si consideri quindi il programma HOFL:

$$t = \textbf{rec}\,.\lambda n.\lambda k.\,\textbf{if}\,k\,\textbf{then}\,1\,\textbf{else if}\,n-k\,\textbf{then}\,1\,\textbf{else}((f\ \ n-1)\ h) + ((f\ \ n-1)\ \ k-1)$$

se ne calcoli il tipo e si valuti a forma canonica il termine $((t\ 2)\ \ 1)$.

Svolgimento L'insieme B è quello delle coppie (n, k) $\quad n \in \omega, 0 \leq k \leq n$. La relazione è semplicemente

$$(n, k) < (n', k') \qquad \text{se } n' = n + 1 \qquad C = \omega$$

La funzione $F((n, k), h)$ è definita per casi su (n, k). Se $k = 0$ oppure $k = n$, F non guarda h e vale $F((n, 0), h) = F((n, n), h) = 1$.
Se $k \neq 0$, n, allora $h : \{(n-1, k')|0 \leq k' \leq n-1\} \to C$. Però $F((n, k), h)$ guarda solo due valori di h, valutati su $(n-1, k-1)$ e $(n-1, k)$. Si ricordi che $k < n$ quindi $k \leq n-1$. Ne conseguono le 3 definizioni date.

$$t = \textbf{rec}\ \underbrace{f}_{int\to int\to int}\ .\lambda \underbrace{n}_{int}.\lambda \underbrace{k}_{int}.\,\textbf{if}\,k\,\textbf{then}\,1\,\textbf{else if}\,n-k\,\textbf{then}\,1\,\textbf{else}((f\ n-1)\ k) + \underbrace{\underbrace{\underbrace{((\underbrace{f}_{int\to\tau}\ \underbrace{n-1}_{int})}_{\tau=int\to\tau'}}_{\tau'=int}\ \underbrace{k-1}_{int})}_{int}$$

$((t\ 2)\ 1) \to m$
$\Longleftarrow \lambda k.\,\textbf{if}\,k\,\textbf{then}\,1\,\textbf{else if}\,n-k\,\textbf{then}\,1\,\textbf{else}((f\ \ n-1)\ k) + ((f\ \ n-1)\ \ k-1)[^2/_n] \to m$
$\Longleftarrow \textbf{if}\,k\,\textbf{then}\,1\,\textbf{else if}\,2-k\,\textbf{then}\,1\,\textbf{else}((f\ \ 2-1)\ k) + ((f\ \ 2-1)\ \ k-1)[^1/_k] \to m$
$\Longleftarrow \textbf{if}\,1\,\textbf{then}\,1\,\textbf{else if}\,2-1\,\textbf{then}\,1\,\textbf{else}((f\ \ 2-1)\ 1) + ((f\ \ 2-1)\ \ 1-1) \to m$
$\Longleftarrow ((t\ 2-1)\ 1) + ((t\ 2-1)\ 1-1) \to m$
$\Longleftarrow (\textbf{if}\,1\,\textbf{then}\,1\,\textbf{else if}(2-1)-1\,\textbf{then}\,1\,\textbf{else}((f\ \ (2-1)-1)\ 1) + ((f\ \ (2-1)-1)\ \ 1-1))+$
$\textbf{if}\,1-1\,\textbf{then}\,1\,\textbf{else if}(2-1)-(1-1)\,\textbf{then}\,1\,\textbf{else}((f\ \ (2-1)-1)\ 1-1) + ((f\ \ (2-1)-1)\ \ (1-1)-1) \to m$
$\overset{m=m_1+m_2}{\Longleftarrow\!=\!=} (2-1)-1 \to 0 \quad 1 \to m_1 \quad 1-1 \to 0 \quad 1 \to m_2$
$\overset{m_1=1 \quad m_2=1}{\Longleftarrow\!=\!=\!=\!=} m = m_1 + m_2 = 1 + 1 = 2$

B.0.12. Correzione esercizi 23/03/2010

B.0.12.1. Esercizio 1

Si consideri il CPO con bottom $(\mathscr{P}(\omega), \subseteq)$ costituito di numeri naturali $S \subseteq \omega$, ordinati per inclusione. Le funzioni $f, g : \mathscr{P}(\omega) \to \mathscr{P}(\omega)$ così definite:

$$f = \lambda S.S \cap X \qquad g = \lambda.(\omega/S) \cap X \qquad \text{dove } X \subseteq \omega$$

Sono monotone? Sono continue? La risposta dipende da X?

Svologimento Verifichiamo la monotonicità di f. Ci chiediamo:

$$S_1 \subseteq S_2 \overset{?}{\Longrightarrow} S_1 \cap X \subseteq S_2 \cap X$$

Supponiamo per ipotesi

$$S_1 \subseteq S_2$$

La proprietà è dimostrata se

$$\forall n \in \mathbb{N} \quad n \in S_1 \cap X \Longrightarrow n \in S_2 \cap X$$

Come al solito, supponiamo sia vera la premessa, dimostriamo la conseguenza. Assumiamo che $n \in S_1 \cap X$. Quandi $n \in S_1 \wedge n \in X$ ma, visto che $S_1 \subseteq S_2$, allora $n \in S_2 \cap X$ $\square$

Dimostriamo la continuità di f. Ci chiediamo

$$n \in \cup_{i\in\omega}(S_i \cap x) \overset{?}{\Longleftrightarrow} n \in (\cup_{i\in\omega} S_i) \cap X$$

$$\begin{aligned} n \in \cup_{i\in\omega}(S_i \cap X) \quad &\Leftrightarrow \quad \exists i.n \in (S_i \cap X) \quad \Leftrightarrow \quad n \in S_i \wedge n \in X \quad \Leftrightarrow \quad \exists i, n.n \in S_i \wedge n \in X \\ &\Leftrightarrow \quad n \in X \wedge \exists i.n \in S_i \quad \Leftrightarrow \quad n \in X \wedge n \in \cup_{i\in\omega} S_i \quad \Leftrightarrow \quad n \in (\cup S_i) \cap X \quad \square \end{aligned}$$

Queste tipo di dimostrazioni funzionano bene con l'appartenenza.

Verifichiamo la monotonicità nel caso della funzione g La cosa dipende da X: Se prendo $X = \emptyset$, ottengo $\forall S_i, \emptyset$, quindi monotona continua.
Se $X \neq \emptyset$, Non potrà essere monotona, basta prendere un controesempio

$$S_1 \subseteq S_2 \quad \lambda \notin S_1 \quad \lambda \in S_2 \quad \lambda \in X$$

$$\lambda \in g(S_1) \quad \lambda \notin g(S_2)$$

B.0.12.2. Esercizio 3

1. Si definisca un ordinamento parziale $\mathscr{D} = (D, \sqsubseteq)$ che non sia completo.

2. L'ordinamento inverso non riflessivo $\mathscr{D}' = (D, \sqsubseteq')$ con $x \sqsubseteq' y$ se e solo se $y \sqsubseteq x$ e $x \neq y$, è una relazione ben fondata?

3. In generale, è possibile che $\mathscr{D}'$ sia ben fondato per qualche $\mathscr{D}$?

Svolgimento : da sistemare Prendiamo $\mathscr{D} = (\omega, \leq)$ come esempio, per avere un'intuizione. La catena dei numeri primi $2 \leq 5 \leq 7 \leq \dots$ non ha maggioranti.

$$\mathscr{D}' = (\omega, \sqsupseteq') \qquad x \sqsupseteq' y \quad \Longleftrightarrow \quad y \sqsubseteq x \land x \neq y$$

Se togliessi l'ipotesi $x \neq y$ l'ordinamento non sarebbe sicuramente una relazione ben fondata (loop). Escludo uguaglianze tra x e y. $2 \sqsupseteq' 5 \sqsupseteq' 7 \dots$ è diventata una catena discendente infinita.
Questo succede per sempre? Si, succede sempre!
Un ordine parziale non completo non ha minimo dei maggioranti. Affinché non sia completo, bisogna che una catena non sia finita, trovo sempre una catena infinita nell'ordinamento $\mathscr{D}$.
In questa sequenza posso usare la relazione $x \leq x$.
Vogliamo vedere se, avendo questa catena infinita, riusciamo a costruire un'altra catena infinita senza ripetizioni (perché ho infiniti elementi diversi, non si stabilizza mai). Quando faccio la catena discendente infinita, non devo avere ripetizioni. L'elemento x non è in relazione con se stesso.
Prendiamo

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$$

catena infinita senza lub. Immagginiamo sia infinita.
In questa catena posso eliminare gli elementi uguali ed ottenere una catena infinita, applicando la seguente funzione:

$$f(i+1) = \min\{j | d_j \neq d_{j-1}, f(i) < j\} \qquad f(0) = 0$$

Cancelliamo gli elementi ripetuti, sarà ancora infinita.

$$d_{f(0)} \sqsubseteq d_{f(1)} \sqsubseteq d_{f(2)} \dots$$

Questa catena non ha elementi ripetuti. Rovesciando diventa decrescente. Il fatto che non ci sono elementi ripetuti ci mette apposto. Abbiamo quindi costriuto una catena discendente infinita senza ripetizioni, cioò dimostra che la relazione non è ben fondata.

B.0.13. Esercizio 4

Si studi l'insieme $V^* \cup V^\infty$ delle stringhe finite (V^*) e infinite V^∞ sull'alfabeto $V = \{a, b, c\}$, con l'ordinamento $\alpha \sqsubseteq \alpha\beta$, dove in $\alpha\beta$ la giustapposizione indica la concatenazioen tra stringhe e $\alpha\beta = \alpha$ se α è infinita

Svolgimento

$$V^* \cup V^\infty \quad V = \{a, b, c\} \quad V^\infty = \omega \to V$$

$$\alpha = a_0 a_1 a_2$$

$$\alpha(0) = a_0 \quad \alpha(1) = a_1 ...$$

$$V^* = \cup V_n$$

Ordinamento:

$$\alpha \sqsubseteq \alpha\beta \quad \alpha_\infty \beta = \alpha_\infty$$

Estesione ai numeri naturale. Invece di una sola operazione di successore, ne abbiamo tante.
Questo è CPO? dobbiamo verificare le seguenti proprietà:

1. Properietà riflessiva.

$$\alpha \sqsubseteq \alpha$$

Se prendo $\beta = \lambda$, otteniamo $\alpha \sqsubseteq \alpha$. Se prendiamo quindi λ, la stringa vuota, siamo a posto.

2. Proprietà antisimmetrica.

$$\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \alpha \stackrel{?}{=} \alpha = \beta$$

Assumiamo le premesse:

$$\beta = \alpha\gamma \quad \alpha = \beta\delta$$

allora

$$\alpha = \alpha\gamma\delta$$

- Se $\alpha \in V^{\infty}$, non c' 'e nessun problema, otteniamo subito $\alpha = \beta$

$$\beta = \alpha \wedge \alpha = \beta$$

La concatenazione col fondo non cambia la stringa

- Se $\alpha \in V^*$

$$\gamma\delta = \lambda \Longrightarrow \gamma = \lambda \quad \delta = \lambda \Longrightarrow \alpha = \beta$$

3. Proprietà transitiva

$$\gamma \sqsubseteq \beta \wedge \beta \sqsubseteq \gamma \stackrel{?}{\Longrightarrow} \alpha \sqsubseteq \gamma$$

$$\beta = \alpha\delta \quad \gamma \sqsubseteq \beta\Sigma \quad \gamma = \alpha\delta\Sigma$$

Dimostriamo che è completo.
Ci chiediamo se esista maggiorante della catena

$$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \qquad \sqcup_{i\in\omega}\, \alpha_i$$

- Catena finita: ho sicuramente miniimo dei maggioranti che è l'elemento sui cui la catena si stabilizza. Naturalmente, in questo caso, comprendo quello in cui ad un certo punto, entra una strin ga $g \in V^*$

- Catena infinita: questo è il caso di catene infinite formate sono da stringhe $\in V^*$, allora se prendiamo n grande a piacere $\exists k.|\alpha_k| \geq n$? Certo, altrimenti sarebbe finita. Ma se io voglio vedere l'ennesimo elemento della stringa infinita, posso usare una approssimazione lunga k ma io so che questo valore ennesimo non sarà mai modificato.

$$(\bigsqcup \alpha_i)(n) = \alpha_{k_n}|n$$

Magia nera.

Esercizio 5 Verificare che i comandi

$$x := 0; \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2 \qquad \text{e} \qquad x := 0; c_1$$

sono semanticamente equivalenti. Si svolga la prova usando sia la operazionale che la semantica denotazionale.

Svolgimento

$$x := 0; \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2 \equiv_{den} x := 0; c_1$$

Dobbiamo far vedere che:

$$\mathscr{C}[\![x := 0; \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2]\!] \equiv_{den} \mathscr{C}[\![x := 0; c_1]\!]$$

$$\mathscr{C}[\![x := 0; \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma = \mathscr{C}[\![\textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2]\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) =$$
$$\mathscr{C}[\![\textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma[0/x] = (\lambda\sigma'.\mathscr{B}[\![x = 0]\!]\sigma' \to \mathscr{C}[\![c_1]\!]\sigma', \mathscr{C}[\![c_2]\!]\sigma')\sigma[0/x] =$$
$$\mathscr{B}[\![x = 0]\!]\sigma[0/x] \to \mathscr{C}[\![c_1]\!]\sigma[0/x], \mathscr{C}[\![c_2]\!]\sigma[0/x] = \underbrace{\mathscr{C}[\![c_1]\!]\sigma[0/x]}$$

$$\mathscr{C}[\![x := 0; c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) = \underbrace{\mathscr{C}[\![c_1]\!]\sigma[0/x]}$$

B.0.13.1. Esercizio 6

Si verifichi l'equivalenza dei comandi

$$\textbf{while } b \textbf{ do } c \qquad \textbf{while } b \textbf{ do if } b \textbf{ then } c \textbf{ else skip}$$

Svolgimento

$$\textbf{while } b \textbf{ do } c \equiv_{den} \textbf{while } b \textbf{ do if } b \textbf{ then } c \textbf{ else skip}$$

$$\mathscr{C}[\![c_1]\!] = fix\Gamma \quad \Gamma\varphi\sigma = \mathscr{B}[\![b]\!]\sigma \to \gamma^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

$$\mathscr{C}[\![c_2]\!] = fix\Gamma' \quad \Gamma'\varphi\sigma = \mathscr{B}[\![b]\!]\sigma \to \gamma^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \sigma), \sigma$$

$$\Gamma = \Gamma'$$

Allora il corrispondente punto fisso deve essere uguale. Come si fa a vedere che i punti fissi sono uguali?

$$\mathscr{B}[\![b]\!]\sigma = \textbf{true} \quad \Gamma\varphi\sigma = \varphi^*(\mathscr{C}[c]\sigma) \quad \Gamma'\varphi\sigma = \varphi^*(\mathscr{C}[\![c]\!]\sigma)$$

$$\mathscr{B}[\![b]\!]\sigma = \textbf{false} \quad \Gamma\varphi\sigma = \sigma \quad \Gamma'\varphi\sigma = \sigma$$

B.0.13.2. Esercizio 7

Si dimostri la seguenze uguaglianza

$$\mathscr{C}[\![\textbf{while true do skip}]\!] = \mathscr{C}[\![\textbf{while true do } x := x + 1]\!]$$

Svolgimento I programmi non terminano mai, mi aspetto che abbiano la stessa semantica.

$$\mathscr{C}[\![c_1]\!] = fix\Gamma \quad \Gamma\varphi\sigma = \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*([\textbf{skip}]\sigma), \sigma$$

Butto via la stella

$$\Gamma\varphi\sigma = \varphi\sigma \quad \text{cioè} \quad \Gamma = \lambda\varphi.\varphi$$

Funzione identità. Minimo punto fisso. Costruzione del punto fisso

$$\varphi_0 = \lambda\sigma.\bot$$

$$\varphi_1 = \Gamma\varphi_0 = \lambda\sigma.\bot$$

Abbiamo già trovato il punto fisso. Vediamo con l'altro comando.

$$\mathscr{C}[\![c_2]\!] = fix\Gamma'$$

$$\Gamma'\varphi\sigma = \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![x = x + 1]\!]\sigma), \sigma = \varphi\sigma[^{\sigma x+1}/_x] = \lambda\varphi\lambda\sigma.\sigma[^{\sigma x+1}/_x]$$

Butto via la stellina.

$$\varphi_0 = \lambda\sigma.\bot_{\sigma_\bot}$$

$$\varphi_1 = \underbrace{(\lambda\varphi.\ \underbrace{\lambda\sigma.\ \underbrace{\varphi\ \underbrace{\sigma[^{\sigma x+1}/_x]}_{\Sigma}}_{\Sigma_\bot}}_{\Sigma\to\Sigma_\bot})}_{(\sigma\to\sigma_\bot)\to(\sigma\to\sigma_\bot)}(\lambda\sigma.\bot_{\Sigma_\bot}) = \lambda\sigma(\lambda\sigma.\bot_{\sigma_\bot})\sigma[^{\sigma x+1}/_x] = \lambda\sigma.\bot_{\Sigma_\bot}$$

Parte VI.

Elenchi

Elenco delle definizioni

1.1. Sistema di regole di inferenza 3
theo.1.1 theo.1.2
1.3. Derivazione 3
theo.1.3
1.4. Teorema 4
theo.1.4
1.5. Insieme dei teoremi in R 4
theo.1.5 theo.1.6 theo.1.7 theo.1.8 theo.1.9 theo.1.10 theo.1.11 theo.1.12 theo.1.13 theo.1.14 theo.1.15 theo.1.16
2.1. Relazione su un insieme A 19
theo.2.1
2.2. Catena discendente infinita 19
theo.2.2
2.3. Relazione ben fondata 19
theo.2.3
2.4. Chiusura transitiva 19
theo.2.4 theo.2.5 theo.2.6 theo.2.7 theo.2.8 theo.2.9 theo.2.10 theo.2.11 theo.2.12 theo.2.13 theo.2.14 theo.2.15
2.16. Termini su una segnatura tipata 23
theo.2.16
2.17. Induzione strutturale 24
theo.2.17 theo.2.18 theo.2.19
2.20. Sottoderivazione immediata 25
theo.2.20 theo.2.21 theo.2.22 theo.2.23 theo.2.24
2.25. Insieme dei predecessori 28
theo.2.25 theo.2.26 theo.2.27 theo.2.28 theo.2.29
3.1. Ordinamento Parziale 31
theo.3.1 theo.3.2 theo.3.3
3.4. Ordinamento totale 32
theo.3.4 theo.3.5 theo.3.6 theo.3.7
3.8. Diagramma di Hasse 33
theo.3.8 theo.3.9
3.10. Chiusura 33
theo.3.10
3.11. Minimo di un insieme 34
theo.3.11 theo.3.12 proof.1
3.13. Elemento Minimale di un insieme 34
theo.3.13
3.14. Maggiorante (upper bound) 34
theo.3.14
3.15. Minimo dei maggioranti (LUB) 34
theo.3.15
3.16. Catena 35
theo.3.16 theo.3.17
3.18. Limite di una catena 36
theo.3.18

3.19. Altezza di un insieme . . . 36
theo.3.19 theo.3.20
3.21. Sottoinsieme diretto . . . 36
theo.3.21
3.22. Insieme inclusivo . . . 36
theo.3.22 theo.3.23 theo.3.24
3.25. Ordinamento parziale completo (CPO) . . . 38
theo.3.25
3.26. Ordinamento parziale completo con bottom . . . 38
theo.3.26 theo.3.27 theo.3.28 theo.3.29 theo.3.30 theo.3.31
3.32. Reticolo . . . 40
theo.3.32 theo.3.33
3.34. Reticolo completo . . . 41
theo.3.34 theo.3.35 notaz.1
3.36. Monotonia . . . 41
theo.3.36 theo.3.37
3.38. Continuità . . . 42
theo.3.38 theo.3.39 theo.3.40 theo.3.41
3.42. Punto fisso . . . 43
theo.3.42
3.43. Punto prefisso . . . 43
theo.3.43 theo.3.44 theo.3.45 theo.3.46 theo.3.47 theo.3.48
3.49. Operatore delle conseguenze immediate ($\hat{R}$) . . . 46
theo.3.49 theo.3.50 theo.3.51 theo.3.52 theo.3.53 theo.3.54 theo.3.55 notaz.2 theo.4.1
4.2. Induzione computazionale . . . 60
theo.4.2 theo.4.3 theo.4.4
5.1. Variabili libere . . . 65
theo.5.1
5.2. Sostituzione . . . 65
theo.5.2
5.3. α-conversione . . . 66
theo.5.3
5.4. β-conversione (copy-rule o β-reduction) . . . 66
theo.5.4 theo.5.5 theo.5.6 theo.5.7 theo.5.8 theo.5.9 theo.5.10
5.11. α-conversione tipata . . . 71
theo.5.11
5.12. Variabili libere in un termine HOFL . . . 71
theo.5.12
5.13. Variabili legate in un termine HOFL . . . 72
theo.5.13
5.14. Forme canoniche . . . 72
theo.5.14 theo.5.15 theo.5.16 theo.5.17 theo.6.1
6.2. Operatore π di proiezione . . . 79
theo.6.2 theo.6.3 theo.6.4 theo.6.5 theo.6.6 theo.6.7 theo.6.8 theo.6.9 theo.6.10 theo.6.11
6.12. Operatore di lifting . . . 84
theo.6.12
6.13. Operatore `let` . . . 84
theo.6.13 theo.6.14 theo.6.15 theo.6.16 theo.6.17 theo.7.1 theo.7.2 theo.7.3 theo.7.4 theo.8.1 theo.8.2
8.3. Collegamento di due agenti . . . 108
theo.8.3
8.4. Agenti Guarded . . . 108
theo.8.4 theo.8.5 theo.8.6 theo.8.7 theo.8.8 theo.8.9 theo.8.10

8.11. Relazione di equivalenza 110
theo.8.11
8.12. Contesto 111
theo.8.12
8.13. Congruenza 111
theo.8.13 theo.8.14
8.15. Strong bisimulation 111
theo.8.15
8.16. Strong bisimilarity $\simeq$ 111
theo.8.16 theo.8.17 theo.8.18 theo.8.19 theo.8.20 theo.8.21 theo.8.22 theo.8.23 theo.8.24 theo.8.25 theo.8.26 theo.8.27
8.28. Satisfaction relation 118
theo.8.28 theo.8.29
8.30. Profondità di una formula 119
theo.8.30 theo.8.31 theo.8.32 theo.8.33 theo.8.34 theo.8.35 theo.8.36 theo.9.1
9.2. Binding occourrence nel Π-calcolo 127
theo.9.2
9.3. Free name 127
theo.9.3
9.4. Bound name 127
theo.9.4 theo.9.5 theo.9.6 theo.9.7 theo.9.8 theo.9.9 theo.9.10 theo.9.11 theo.9.12
9.13. Sostituzione 134
theo.9.13
9.14. Applicazione di sostituzione 135
theo.9.14
9.15. Bisimilarità early non ground 135
theo.9.15
9.16. Bisimilarità late non ground 135
theo.9.16
10.1. Sigma Field (σ-algebra) 139
theo.10.1 theo.10.2
10.3. Misura su (Ω, F) 139
theo.10.3
10.4. Sub-probability 140
theo.10.4
10.5. Probability 140
theo.10.5 theo.10.6
10.7. Topologia 140
theo.10.7
10.8. Topologia euclidea 141
theo.10.8 theo.10.9 theo.10.10 theo.10.11 theo.10.12
10.13Catena di Markov 143
theo.10.13
10.14Catena di Markov omogenea 143
theo.10.14
10.15Discrete-time Markov Chain (DTMC) 144
theo.10.15
10.16Continuous-time Markov Chain (CTMC) 144
theo.10.16 theo.10.17
10.18Probabilità di un cammino 146
theo.10.18 theo.10.19 theo.10.20
10.21Catene ergodiche 146

theo.10.21 theo.10.22 theo.10.23 theo.11.1 theo.11.2 theo.11.3 theo.11.4 theo.11.5 theo.11.6

Elenco dei teoremi

theo.1.1 theo.1.2 theo.1.3 theo.1.4 theo.1.5 theo.1.6 theo.1.7 theo.1.8 theo.1.9 theo.1.10 theo.1.11 theo.1.12 theo.1.13 theo.1.14 theo.1.15 theo.1.16 theo.2.1 theo.2.2 theo.2.3 theo.2.4
2.5. Proprietà delle relazioni ben fondate 20
theo.2.5
2.6. Relazione ben fondata su insiemi finiti 20
theo.2.6 theo.2.7 theo.2.8 theo.2.9 theo.2.10 theo.2.11
2.12. Fondatezza della chiusura transitiva di $\prec$ 22
theo.2.12
2.13. 22
theo.2.13
2.14. Induzione matematica debole 23
theo.2.14
2.15. Induzione matematica forte 23
theo.2.15 theo.2.16 theo.2.17 theo.2.18 theo.2.19 theo.2.20 theo.2.21 theo.2.22 theo.2.23 theo.2.24 theo.2.25
2.26. Ricorsione ben fondata 28
theo.2.26 theo.2.27 theo.2.28 theo.2.29 theo.3.1 theo.3.2 theo.3.3 theo.3.4 theo.3.5 theo.3.6
3.7. Sottoinsiemi di un insieme totalmente ordinato 33
theo.3.7 theo.3.8
3.9. Equivalenza fra relazione e chiusura di Hasse 33
theo.3.9 theo.3.10 theo.3.11
3.12. Unicità del minimo 34
theo.3.12 proof.1 theo.3.13 theo.3.14 theo.3.15 theo.3.16
3.17. Catene di un insieme finito 35
theo.3.17 theo.3.18 theo.3.19 theo.3.20 theo.3.21 theo.3.22 theo.3.23
3.24. Catene di insiemi parzialmente ordinati 38
theo.3.24 theo.3.25 theo.3.26 theo.3.27 theo.3.28 theo.3.29 theo.3.30 theo.3.31 theo.3.32 theo.3.33 theo.3.34 theo.3.35 notaz.1 theo.3.36 theo.3.37 theo.3.38 theo.3.39
3.40. Relazione fra monotonia e continuità (1) 42
theo.3.40
3.41. Relazione fra monotonia e continuità (2) 43
theo.3.41 theo.3.42 theo.3.43 theo.3.44
3.45. Teorema del punto fisso (o teorema di Kleene) 44
theo.3.45 theo.3.46 theo.3.47
3.48. Teorema di Tarski 46
theo.3.48 theo.3.49
3.50. Catena di applicazioni $\hat{R}(\perp)$ 47
theo.3.50
3.51. Monotonia di $\hat{R}$ 47
theo.3.51
3.52. Continuità di $\hat{R}$ 47
theo.3.52
3.53. Punto fisso di $\hat{R}$ 48
theo.3.53 theo.3.54 theo.3.55 notaz.2 theo.4.1 theo.4.2 theo.4.3 theo.4.4 theo.5.1 theo.5.2 theo.5.3 theo.5.4 theo.5.5 theo.5.6 theo.5.7 theo.5.8 theo.5.9
5.10. La sostituzione rispetta i tipi 71

theo.5.10 theo.5.11 theo.5.12 theo.5.13 theo.5.14 theo.5.15 theo.5.16
5.17. 74
theo.5.17
6.1. Completezza del $CPO_\perp$ per $\tau ::= \tau \times \tau$ 78
theo.6.1 theo.6.2
6.3. Continuità di π 79
theo.6.3 theo.6.4
6.5. Completezza dello spazio funzionale 80
theo.6.5
6.6. Continuità di (f,g) 80
theo.6.6
6.7. Continuità delle singole funzioni di una coppia 81
theo.6.7 theo.6.8
6.9. Continuità di una funzione con una coppia come argomento 82
theo.6.9 theo.6.10 theo.6.11 theo.6.12 theo.6.13
6.14. Continuità di $\underline{op}$ 86
theo.6.14
6.15. Continuità di Cond 87
theo.6.15
6.16. Continuità della semantica della lambda expression 89
theo.6.16
6.17. Lemma di sostituzione 90
theo.6.17 theo.7.1
7.2. Se t converge operazionalmente allora converge anche denotazionalmente 96
theo.7.2
7.3. Uguaglianza fra semantiche per termini di tipo *int* 97
theo.7.3
7.4. Divergenza fra semantiche per termini di tipo composto 98
theo.7.4 theo.8.1 theo.8.2 theo.8.3 theo.8.4 theo.8.5 theo.8.6 theo.8.7 theo.8.8 theo.8.9 theo.8.10 theo.8.11 theo.8.12 theo.8.13 theo.8.14 theo.8.15 theo.8.16
8.17. 112
theo.8.17 theo.8.18
8.19. Bisimulazione come punto fisso 113
theo.8.19
8.20. Bisimilarità come massimo punto fisso 113
theo.8.20 theo.8.21
8.22. Monotonia di Φ 114
theo.8.22 theo.8.23
8.24. 114
theo.8.24 theo.8.25 theo.8.26 theo.8.27 theo.8.28 theo.8.29 theo.8.30
8.31. 119
theo.8.31 theo.8.32 theo.8.33 theo.8.34 theo.8.35 theo.8.36 theo.9.1 theo.9.2 theo.9.3 theo.9.4 theo.9.5 theo.9.6 theo.9.7 theo.9.8 theo.9.9 theo.9.10 theo.9.11 theo.9.12 theo.9.13 theo.9.14 theo.9.15 theo.9.16 theo.10.1 theo.10.2 theo.10.3 theo.10.4 theo.10.5 theo.10.6 theo.10.7 theo.10.8 theo.10.9
10.10La distribuzione esponenziale è senza memoria 142
theo.10.10
10.11Transizioni multiple 142
theo.10.11
10.12Confronto di due eventi 142
theo.10.12 theo.10.13 theo.10.14 theo.10.15 theo.10.16 theo.10.17 theo.10.18 theo.10.19 theo.10.20 theo.10.21 theo.10.22 theo.10.23 theo.11.1 theo.11.2
11.3. Bisimilarità e formule di Larsen-Skou 152

theo.11.3 theo.11.4 theo.11.5
11.6. Segala è più espressivo di Generative 154
theo.11.6

Elenco degli esempi

theo.1.1
1.2. Regola di inferenza 3
theo.1.2 theo.1.3 theo.1.4 theo.1.5
1.6. Grammatiche viste come sistemi di regole 4
theo.1.6
1.7. Formule valide e non 7
theo.1.7
1.8. Dimostrazione di appartenenza ad IMP 7
theo.1.8
1.9. Valutazione di un comando tramite la semantica operazionale 10
theo.1.9
1.10. Uguaglianza fra comandi (1) 11
theo.1.10
1.11. Uguaglianza fra comandi (2) 12
theo.1.11
1.12. Dimostrazioni parametriche (1) 13
theo.1.12
1.13. Dimostrazioni parametriche (2) 14
theo.1.13
1.14. Dimostrazioni di non–terminazione 14
theo.1.14
1.15. Dimostrazioni di non–uguaglianza 16
theo.1.15
1.16. Estensione della grammatica di Aexpr 17
theo.1.16 theo.2.1 theo.2.2 theo.2.3 theo.2.4 theo.2.5 theo.2.6 theo.2.7
2.8. Numeri Naturali 21
theo.2.8
2.9. Termini e sottotermini 21
theo.2.9
2.10. Ordinamento Lessicografico 21
theo.2.10
2.11. Numeri interi 22
theo.2.11 theo.2.12 theo.2.13 theo.2.14 theo.2.15 theo.2.16 theo.2.17
2.18. Segnatura in IMP 24
theo.2.18
2.19. Induzione strutturale 24
theo.2.19 theo.2.20
2.21. Sottoderivazione immediata 25
theo.2.21
2.22. Sottoderivazione propria 25
theo.2.22
2.23. Dimostrazione mediante induzione sulle derivazioni 25
theo.2.23
2.24. Ricorsione ben fondata 27
theo.2.24 theo.2.25 theo.2.26
2.27. Formalizzazione della ricorsione ben fondata 28

theo.2.27
2.28. Ricorsione ben fondata per una visita di albero in Lisp 28
theo.2.28
2.29. Funzione di Ackermann 29
theo.2.29 theo.3.1
3.2. Ordinamento parziale 32
theo.3.2
3.3. Ordinamento non parziale 32
theo.3.3 theo.3.4
3.5. Ordinamento discreto 32
theo.3.5
3.6. Ordinamento piatto 32
theo.3.6 theo.3.7 theo.3.8 theo.3.9 theo.3.10 theo.3.11 theo.3.12 proof.1 theo.3.13 theo.3.14 theo.3.15 theo.3.16 theo.3.17 theo.3.18 theo.3.19
3.20. Insieme infinito con catene finite 36
theo.3.20 theo.3.21 theo.3.22
3.23. Fairness, insieme non inclusivo 37
theo.3.23 theo.3.24 theo.3.25 theo.3.26
3.27. Ordinamento parziale non completo (1) 38
theo.3.27
3.28. Ordinamento parziale completo (2) 38
theo.3.28
3.29. Powerset con inclusione 39
theo.3.29
3.30. Funzioni parziali 39
theo.3.30
3.31. Sequenza di funzioni 40
theo.3.31 theo.3.32
3.33. Reticolo 41
theo.3.33 theo.3.34
3.35. Reticolo non completo 41
theo.3.35 notaz.1 theo.3.36
3.37. Funzione non monotona 42
theo.3.37 theo.3.38
3.39. Funzione monotona non continua 42
theo.3.39 theo.3.40 theo.3.41 theo.3.42 theo.3.43 theo.3.44 theo.3.45
3.46. $\perp$ necessario 45
theo.3.46
3.47. Continuità necessaria 45
theo.3.47 theo.3.48 theo.3.49 theo.3.50 theo.3.51 theo.3.52 theo.3.53
3.54. Insieme di regole con $\hat{R}$ non continuo 49
theo.3.54
3.55. Stringhe di una grammatica 50
theo.3.55 notaz.2
4.1. Semantica denotazionale di un comando 54
theo.4.1 theo.4.2
4.3. Esempio 60
theo.4.3
4.4. Applicazione induzione computazionale 2 61
theo.4.4 theo.5.1 theo.5.2 theo.5.3 theo.5.4
5.5. looping term 66
theo.5.5

5.6. Y combinator 67
theo.5.6
5.7. Programma tipabile (Church) 68
theo.5.7
5.8. Programma non tipabile (Curry) 69
theo.5.8
5.9. Programma non tipabile 70
theo.5.9 theo.5.10 theo.5.11 theo.5.12 theo.5.13 theo.5.14
5.15. 74
theo.5.15
5.16. Esempio di valutazione eager e lazy 74
theo.5.16 theo.5.17 theo.6.1 theo.6.2 theo.6.3 theo.6.4 theo.6.5 theo.6.6 theo.6.7 theo.6.8 theo.6.9
6.10. Continuità della funzione apply 82
theo.6.10
6.11. Funzioni Curry ed Uncurry 83
theo.6.11 theo.6.12 theo.6.13 theo.6.14 theo.6.15 theo.6.16 theo.6.17
7.1. Distinzione dei $\perp$ 94
theo.7.1 theo.7.2 theo.7.3 theo.7.4
8.1. Algebre di processo 103
theo.8.1
8.2. Esempio di derivazione 107
theo.8.2 theo.8.3 theo.8.4 theo.8.5
8.6. Agente non guardato 108
theo.8.6
8.7. Agenti isomorfi 109
theo.8.7
8.8. Grafi differenti, stringhe uguali, comportamento uguale 109
theo.8.8
8.9. 110
theo.8.9
8.10. Grafi differenti, stringhe uguali, comportamento differente 110
theo.8.10 theo.8.11 theo.8.12 theo.8.13
8.14. Teoria dei giochi per l'esempio 8.10 111
theo.8.14 theo.8.15 theo.8.16 theo.8.17
8.18. Stati bisimili 112
theo.8.18 theo.8.19 theo.8.20
8.21. Bisimilarità come massimo punto fisso 113
theo.8.21 theo.8.22 theo.8.23 theo.8.24
8.25. Calcolo Bisimilarità 115
theo.8.25
8.26. Punto fisso alla prima iterazione 115
theo.8.26
8.27. Equivalenza a stringhe non è congruenza 116
theo.8.27 theo.8.28
8.29. Agenti non bisimilari tramite formule logiche 118
theo.8.29 theo.8.30 theo.8.31
8.32. 120
theo.8.32
8.33. weak bisimilarity non è congruenza 120
theo.8.33
8.34. Weak Congruence non è bisimulazione 121
theo.8.34

8.35. 122
theo.8.35
8.36. Esempio 8.34 con dynamic 122
theo.8.36
9.1. Telefoni mobili 125
theo.9.1 theo.9.2 theo.9.3 theo.9.4
9.5. Violazione della regola dei nomi in input 128
theo.9.5
9.6. Utilizzo del comando di matching 129
theo.9.6
9.7. Violazione della regola sui nomi nel parallelo 130
theo.9.7
9.8. Estrusione di variabile 131
theo.9.8
9.9. Derivazione nel Π-calcolo (1) 132
theo.9.9
9.10. Derivazione nel Π-calcolo (2) 132
theo.9.10
9.11. Bisimularità early e late 133
theo.9.11
9.12. La bisimilarità ground non è una congruenza 134
theo.9.12 theo.9.13 theo.9.14 theo.9.15 theo.9.16 theo.10.1
10.2. 139
theo.10.2 theo.10.3 theo.10.4 theo.10.5
10.6. Cammini finiti 140
theo.10.6 theo.10.7 theo.10.8
10.9. Distribuzione senza memoria 141
theo.10.9 theo.10.10 theo.10.11 theo.10.12 theo.10.13 theo.10.14 theo.10.15 theo.10.16
10.17DTMC 145
theo.10.17 theo.10.18
10.19Probabilità di un cammino 146
theo.10.19
10.20Random walk 146
theo.10.20 theo.10.21
10.22Punto di stabilizzazione 147
theo.10.22
10.23. 148
theo.10.23
11.1. Reactive probabilistic transition system 151
theo.11.1
11.2. Logica di Larsen-Skou 152
theo.11.2 theo.11.3
11.4. Segala automaton 153
theo.11.4
11.5. Simple Segala Automaton 154
theo.11.5 theo.11.6