UNIVERSITÀ DEGLI STUDI FIRENZE

Modelli di Sistemi Sequenziali e Concorrenti

Esercizi per l'esame di MSSC

Anno Accademico 2015/2016

Tommaso Papini
tommaso.papini1@stud.unifi.it
5537529

Esercizio 3.9

Scrivere una grammatica dipendente da contesto per il linguaggio $L \triangleq \{a^n b a^n c a^n | n \geq 1\}$.

La grammatica G che genera il linguaggio L richiesto è una quadrupla $G \triangleq < A, V, S, P >$. Seguendo la convenzione nota per rappresentare le grammatiche in forma compatta[1], proponiamo di seguito le produzioni della grammatica G di interesse:

$$S ::= AF \quad (1)$$
$$A ::= abTC | aABC \quad (2)$$
$$TF ::= Tc \quad (3)$$
$$Tc ::= ac \quad (4)$$

$$swap(C, B) \quad (5)$$
$$swap(T, B) \quad (6)$$
$$swap(C, F) \quad (7)$$

$$bB ::= ba \quad (8)$$
$$aB ::= aa \quad (9)$$
$$cC ::= ca \quad (10)$$
$$aC ::= aa \quad (11)$$

L'idea generale è quella di usare due nonterminali segnaposto B e C che, intuitivamente, rappresentano i simboli a dopo la b e dopo la c, rispettivamente. Per riordinare le B e le C e farle andare nella loro posizione finale (ovvero prima tutte le B e poi tutte le C) si usano due nonterminali delimitatori, T ed F: la T rappresenta l'ultima a dopo la b e serve a spostare tutte le B verso sinistra, mentre la F rappresenta il simbolo c ed ha lo scopo di spostare tutte le C verso destra. Una volta che le B risulteranno a sinistra della T, esse saranno nella loro posizione finale e potranno trasformarsi in una a (regole (8) e (9). Analogamente per le C, con la differenza che per trasformarsi in a avranno bisogno del simbolo c (regola (10)), che comparirà solo dopo aver messo tutte le B e le C in ordine (ovvero quando i delimitatori T ed F si toccano, regola (3)). Una volta riordinate tutte le B e le C, esse potranno essere trasformate in a (si osservi che le B possono essere trasformate anche prima di questo punto, ma questo non ci disturba), mentre i delimitatori T ed F potranno essere trasformati in a e c, rispettivamente (regole (3) e (4).

Le regole (5), (6) e (7) servono, intuitivamente, a scambiare nonterminali tra loro, sfruttando la potenza delle *grammatiche context-sensitive* (questo trucco non sarebbe infatti possibile

[1] Le lettere minuscole indicano simboli terminali, le maiuscole indicano nonterminali ed il nonterminale nel lato sinistro della prima produzione rappresenta il simbolo iniziale (solitamente S).

usando *grammatiche context-free*). La regola (5) serve a riordinare le coppie CB in BC, mettendole nell'ordine corretto. La regola (6) serve a "mettere al sicuro" una B ogni volta che viene trovata dal delimitatore T. Analogamente per la regola (7).

Le regole dalla (5) alla (7) sono state proposte in questa forma compatta per migliorare la chiarezza e la leggibilità dell'insieme di produzioni della grammatica. Di seguito, la loro implementazione formale con grammatiche context-sensitive:

$$\begin{aligned} &swap(C,B) = \{ \\ &\qquad CB ::= XB \\ &\qquad XB ::= XY \\ &\qquad XY ::= BY \\ &\qquad BY ::= BC \\ &\} \end{aligned} \tag{12}$$

$$\begin{aligned} &swap(T,B) = \{ \\ &\qquad TB ::= MB \\ &\qquad MB ::= MN \\ &\qquad MN ::= BN \\ &\qquad BN ::= BT \\ &\} \end{aligned} \tag{13}$$

$$\begin{aligned} &swap(C,F) = \{ \\ &\qquad CF ::= IF \\ &\qquad IF ::= IJ \\ &\qquad IJ ::= FJ \\ &\qquad FJ ::= FC \\ &\} \end{aligned} \tag{14}$$

Esercizio 3.15

Dimostrare che, per ogni espressione aritmetica E descritta nella Sezione 3.3,

$$E \to E_1, E \to E_2 \text{ implica } E_1 \twoheadrightarrow n, E_2 \twoheadrightarrow n.$$

Quello che si richiede in questo esercizio è di dimostrare che quando una stessa espresione aritmetica E (generata tramite la grammatica in Tabella 3.3) viene *computata* in due modi distinti, producendo le nuove espressioni E_1 ed E_2, allora possiamo dire che queste due nuove espressioni vengono *valutate* entrambe lo stesso numero n.

Ragioniamo per induzione sul numero di operatori in E:

- **Caso base:**
  Supponiamo che in E ci sia un solo operatore (se non ci fossero operatori allora E sarebbe composto da un solo numero e non potrebbe essere computato in un'altra espressione, ma

soltanto valutato), allora l'unica regola applicabile è *(op)*. Dal fatto che l'unica regola applicabile è *(op)*, segue che $E_1 = E_2 = n$, per un n opportuno. Allora per la prima regola in Tabella 3.5 si ottiene banalmente:

$$E_1 = n \twoheadrightarrow n$$

$$E_2 = n \twoheadrightarrow n$$

- **Passo induttivo:**
  Supponiamo che in E ci siano $k+1$ operatori, con $k \geq 1$, e che la proposizione valga per tutte le espressioni con k operatori. Allora E sarà della forma $E = E_a \; op \; E_b$, per E_a, E_b e op opportuni, e potrà essere computata tramite *(redl)* oppure *(redr)*.

  Se E_1 ed E_2 sono state computate tramite la stessa regola, allora per quanto detto prima esse risultano uguali e banalmente vengono valutate nello stesso numero.
  Supponiamo allora di aver ottenuto E_1 tramite *(redl)* e E_2 tramite *(redr)*. Questo significa che se $E_a \to E'_a$ ed $E_b \to E'_b$, allora E_1 ed E_2 hanno la forma:

$$E_1 = E'_a \; op \; E_b$$

$$E_2 = E_a \; op \; E'_b$$

  Possiamo osservare come ogni computazione tramite le tre regole fornite implichi sempre il decremento unitario del numero di operatori rispetto all'espressione originale. Ovvero ogni regola elimina sempre uno ed un solo operatore. Questo implica che la proposizione risulta vera, per ipotesi induttiva, per E_1, E_2, E_a, E_b, E'_a e E'_b. Questo ci assicura che, indipendentemente da quali regole di computazione verranno utilizzate, queste espressioni verranno valutate sempre nello stesso numero.
  Possiamo quindi dire che $E'_a \twoheadrightarrow x$ e $E'_b \twoheadrightarrow y$, per certi x e y.

  Sfruttando adesso l'equivalenza tra semantica di computazione e semantica di valutazione[2] notiamo che, avendo $E'_a \twoheadrightarrow x$, allora $E_a \xrightarrow{*} x$. Ricordiamo anche che per definizione $E_a \to E'_a$, ovvero $E_a \xrightarrow{*} x$ e quindi, applicando nuovamente l'equivalenza tra le due semantiche, otteniamo che $E_a \twoheadrightarrow x$. Ovvero sia E_a che E'_a vengono valutate entrambe x.
  Lo stesso discorso si applica ad E_b e E'_b, ricavando che entrambi sono valutati in y.

  Sia quindi $x \; op \; y = n$, allora

$$\frac{E'_a \twoheadrightarrow x \qquad E_b \twoheadrightarrow y}{E'_a \; op \; E_b \twoheadrightarrow n} \; (x \; op \; y = n)$$

$$\frac{E_a \twoheadrightarrow x \qquad E'_b \twoheadrightarrow y}{E_a \; op \; E'_b \twoheadrightarrow n} \; (x \; op \; y = n)$$

  da cui la tesi ricordando la forma di E_1 ed E_2.

  Se, alternativamente, E_1 fosse stato ottenuto tramite *(redr)* ed E_2 tramite *(redl)* i passaggi sarebbero stati del tutto analoghi.

Esercizio 4.7

Costruire gli automi associati alle espressioni regolari $a; b + c$ e $a; b + a; c$.

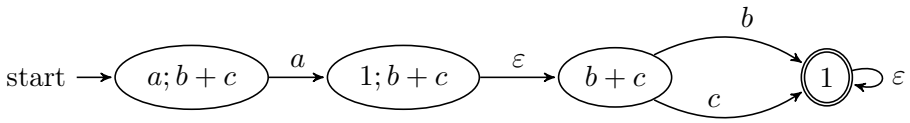

Figura 1: Automa a stati finiti dell'espressione regolare $a; b + c$.

L'automa a stati finiti relativo all'espressione regolare $a; b + c$ è rappresentato in Figura 1. Le transizioni dell'automa in Figura 1 sono giustificate dalle seguenti derivazioni:

$$\dfrac{\dfrac{}{a \xrightarrow{a} 1}\,(Atom)}{a; b + c \xrightarrow{a} 1; b + c}\,(Seq1)$$

$$\dfrac{\dfrac{}{1 \xrightarrow{\varepsilon} 1}\,(Tic)}{1; b + c \xrightarrow{\varepsilon} b + c}\,(Seq2)$$

$$\dfrac{\dfrac{}{b \xrightarrow{b} 1}\,(Atom)}{b + c \xrightarrow{b} 1}\,(Sum1)$$

$$\dfrac{\dfrac{}{c \xrightarrow{c} 1}\,(Atom)}{b + c \xrightarrow{c} 1}\,(Sum2)$$

$$\dfrac{}{1 \xrightarrow{\varepsilon} 1}\,(Tic)$$

Vediamo adesso in Figura 2 l'automa relativo all'espressione $a; b + a; c$.

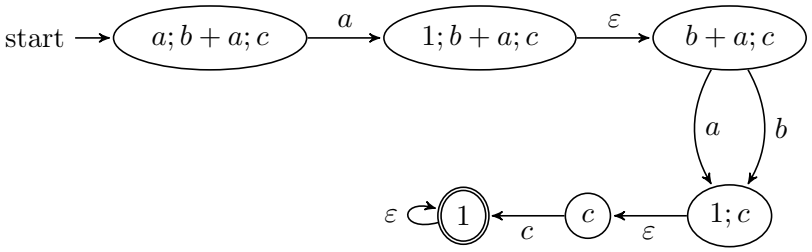

Figura 2: Automa a stati finiti dell'espressione regolare $a; b + a; c$.

Come prima, proponiamo di seguito le derivazioni che giustificano le transizioni del secondo automa:

[2] Questo risultato verrà dato per buono ai fini di questo esercizio in quanto la sua dimostrazione è già obiettivo di un altro esercizio, l'Esercizio 3.14.

$$\dfrac{\dfrac{}{a \xrightarrow{a} 1}\,(Atom)}{a; b + a; c \xrightarrow{a} a; b + a; c}\,(Seq1)$$

$$\dfrac{\dfrac{}{1 \xrightarrow{\varepsilon} 1}\,(Tic)}{1; b + a; c \xrightarrow{\varepsilon} b + a; c}\,(Seq2)$$

$$\dfrac{\dfrac{\dfrac{}{b \xrightarrow{b} 1}\,(Atom)}{b + a \xrightarrow{b} 1}\,(Sum1)}{b + a; c \xrightarrow{b} 1; c}\,(Seq1)$$

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 1}\,(Atom)}{b + a \xrightarrow{a} 1}\,(Sum2)}{b + a; c \xrightarrow{a} 1; c}\,(Seq1)$$

$$\dfrac{\dfrac{}{1 \xrightarrow{\varepsilon} 1}\,(Tic)}{1; c \xrightarrow{\varepsilon} c}\,(Seq2)$$

$$\dfrac{}{c \xrightarrow{c} 1}\,(Atom)$$

$$\dfrac{}{1 \xrightarrow{\varepsilon} 1}\,(Tic)$$

Esercizio 4.10

Si dimostrino le seguenti uguaglianze usando il sistema di inferenza della semantica assiomatica:

$c)\ E = F \Longrightarrow E^* = F^*$
$d)\ E^* + 1 = E^*$
$e)\ E^* + E = E^*$

Sfruttando gli assiomi e le due regole in Tabella 4.4 delle dispense, si dimostrano le uguaglianze proposte.

$c)\ E = F \Longrightarrow E^* = F^*$

Come già mostrato nelle dispense, si può dimostrare che l'uguaglianza $=$ è riflessiva e simmetrica:

$$\dfrac{E + E = E \qquad E + E = E}{E = E \qquad E = E + E}\,(regola1) \tag{15}$$

$$\dfrac{E = F \qquad E = F}{F = F \qquad F = E}\,(regola1) \tag{16}$$

Da qui in avanti useremo i risultati della (15) e della (16) come fossero regole, indicandole con $(rifl)$ e $(simm)$, rispettivamente.

Per dimostrare quindi che $E = F \Longrightarrow E^* = F^*$ basta applicare la regola della sostituzione assumendo riflessività e simmetria della relazione $=$.

$$\dfrac{\dfrac{E = F \qquad \dfrac{}{E^* = E^*}\,(rifl)}{F^* = E^*}\,(regola1)}{E^* = F^*}\,(simm) \tag{17}$$

$d)\ E^* + 1 = E^*$

Dimostriamo innanzitutto la sostitutività rispetto alla somma (che indicheremo con $(sost+)$)

come suggerito dalla Proposizione 4.13 delle dispense:

$$\dfrac{\dfrac{G = H \qquad \dfrac{E = F \qquad \overline{E + G = E + G}\ (rifl)}{F + G = E + G}\ (regola1)}{F + H = E + G}\ (regola1)}{E + G = F + H}\ (simm) \tag{18}$$

La dimostrazione che $E^* + 1 = E^*$ si ottiene quindi come segue (abbiamo diviso la dimostrazione in quattro parti per questioni di leggibilità):

$$\dfrac{\dfrac{\overline{E^* = 1 + E^* E}\ (unfolding) \qquad \overline{1 = 1}\ (rifl)}{E^* + 1 = 1 + E^* E + 1}\ (sost+)}{1 + E^* E + 1 = E^* + 1}\ (simm) \tag{19}$$

$$\dfrac{\overline{1 + E^* E + 1 = 1 + 1 + E^* E}\ (comm+) \qquad \overline{1 + E^* E + 1 = E^* + 1}\ (19)}{1 + 1 + E^* E = E^* + 1}\ (regola1) \tag{20}$$

$$\dfrac{\overline{1 + 1 = 1}\ (idem+) \qquad \overline{1 + 1 + E^* E = E^* + 1}\ (20)}{1 + E^* E = E^* + 1}\ (regola1) \tag{21}$$

$$\dfrac{\dfrac{\dfrac{\overline{E^* = 1 + E^* E}\ (unfolding)}{1 + E^* E = E^*}\ (simm) \qquad \overline{1 + E^* E = E^* + 1}\ (21)}{E^* = E^* + 1}\ (regola1)}{E^* + 1 = E^*}\ (simm) \tag{22}$$

In parole povere, si è fatto un primo passo di unfolding di E^*, ci si è sommato 1 e si è fatto vedere che i due 1 si riducono ad un solo 1. Quindi abbiamo rimesso le cose a posto per avere la formulazione finale richiesta.

e) $E^* + E = E^*$

L'idea di quest'ultima dimostrazione è sulla falsariga della dimostrazione precedente, ovvero si dimostra facendo un passo di unfolding e poi rimettendo tutto a posto. Ad un certo punto sarà molto utile anche sfruttare il risultato della dimostrazione precedente.

$$\dfrac{\dfrac{\overline{E^* = 1 + E^* E}\ (unfolding) \qquad \dfrac{\overline{1E = E}\ (neutro;)}{E = 1E}\ (simm)}{E^* + E = 1 + E^* E + 1E}\ (sost+)}{1 + E^* E + 1E = E^* + E}\ (simm) \tag{23}$$

$$\dfrac{\overline{E^* E + 1E = (E^* + 1)E}\ (distribD) \qquad \overline{1 + E^* E + 1E = E^* + E}\ (23)}{1 + (E^* + 1)E = E^* + E}\ (regola1) \tag{24}$$

$$\dfrac{\overline{E^* + 1 = E^*}\ (22) \qquad \overline{1 + (E^* + 1)E = E^* + E}\ (24)}{1 + E^* E = E^* + E}\ (regola1) \tag{25}$$

$$\dfrac{\dfrac{\dfrac{\overline{E^* = 1 + E^* E}\ (unfolding)}{1 + E^* E = E^*}\ (simm) \qquad \overline{1 + E^* E = E^* + E}\ (25)}{E^* = E^* + E}\ (regola1)}{E^* + E = E^*}\ (simm) \tag{26}$$

Esercizio 7.6

Fornire semantica operazionale e denotazionale del programma

$$\textbf{letrec } f(x) \Leftarrow f(x) \textbf{ in } f(5).$$

Informalmente, possiamo già intuire che il programma presentato diverge, in quanto, indipendentemente dall'argomento passato, l'unica funzione definita $f(x)$ non fa altro che richiamare ricorsivamente se stessa, senza possibilità di arresto.

In particolare, definendo l'insieme di dichiarazioni $D = \{f(x) \Leftarrow f(x)\}$, la *semantica operazionale* con *call-by-name* applicherà un numero infinito di volte la clausola (Fun) in Tabella 7.2:

$$f(5) \xrightarrow{\text{(Fun)}}_D f(5) \xrightarrow{\text{(Fun)}}_D \dots f(5) \xrightarrow{\text{(Fun)}}_D \dots .$$

La semantica operazionale con *call-by-value* risulterà del tutto analoga, in quanto i termini presenti che rappresentano la chiamata a una funzione sono già invocati su rappresentazioni di naturali (in particolare del numero 5) e quindi verrà applicata la clausola (Fun') in Tabella 7.3 all'infinito:

$$f(5) \bullet\!\!\xrightarrow{\text{(Fun')}}_D f(5) \bullet\!\!\xrightarrow{\text{(Fun')}}_D \dots f(5) \bullet\!\!\xrightarrow{\text{(Fun')}}_D \dots .$$

Vediamo adesso come si ricava il significato del programma attraverso la *semantica denotazionale*, ricordando la definizione della funzione $\Omega \equiv \lambda x.\bot$:

$\mathcal{P}[\![\textbf{letrec } f(x) \Leftarrow f(x) \textbf{ in } f(5)]\!]$	
$= \mathcal{T}[\![f(5)]\!]\mathcal{D}[\![f(x) \Leftarrow f(f)]\!]0$	per la clausola (Prg)
$= \mathcal{T}[\![f(5)]\!]fix(\lambda f.\mathcal{T}[\![f(x)]\!]f)0$	per la clausola (Dec)
$= \mathcal{T}[\![f(5)]\!]fix(\lambda f.(\lambda\overline{f}.\lambda x.\overline{f}(\mathcal{T}[\![x]\!]\overline{f}x)f))0$	per la clausola (Fun)
$= \mathcal{T}[\![f(5)]\!]fix(\lambda f.(\lambda\overline{f}.\lambda x.\overline{f}((\lambda\hat{f}.\lambda\hat{x}.x)\overline{f}x)f))0$	per la clausola (Var)
$= \mathcal{T}[\![f(5)]\!]fix(\lambda f.(\lambda\overline{f}.\lambda x.\overline{f}(x)f))0$	per β-riduzione
$= \mathcal{T}[\![f(5)]\!]fix(\lambda f.\lambda x.f(x))0$	per β-riduzione
$= \mathcal{T}[\![f(5)]\!]\ \textbf{sup}\{\lambda f.f^i\Omega \mid i \in \mathbb{N}\}0$	per definizione di minimo punto fisso
$= \mathcal{T}[\![f(5)]\!]\Omega 0$	essendo $(\lambda f.(\lambda x.fx))\Omega = \Omega$[3]
$= (\lambda f.\lambda x.f(\mathcal{T}[\![5]\!]fx))\Omega 0$	per la clausola (Fun)
$= (\lambda f.\lambda x.f((\lambda\hat{f}.\lambda\hat{x}.5)fx))\Omega 0$	per la clausola (Nat)
$= (\lambda f.\lambda x.f(5)\Omega 0$	per β-riduzione
$= \Omega(5)$	per β-riduzione
$= (\lambda x.\bot)5$	per definizione di Ω
$= \bot$	per β-riduzione

Grazie alla semantica denotazionale abbiamo quindi una conferma più formale della divergenza del programma. Infatti interpretare un programma come $\bot$ equivale a dire che esso è privo di significato.

[3] Ovvero, il funzionale su f applicato ad Ω da sempre Ω, il che rende Ω un punto fisso. Equivale a concludere che l'applicazione di una f non rende le approssimazioni successive più definite, in quanto non aggiunge informazione sul significato della funzione.

Esercizio 8.6

Si calcoli formalmente, usando la semantica denotazionale di TINY, l'insieme degli stati a partire dai quali eseguendo il comando seguente si ha divergenza:

while $x > 0$ **do**
if $x < 3$ **then** $x := x - 1$ **else noaction**.

Per semplicità, analizziamo le interpretazioni semantiche di espressioni e comandi che formano il comando while proposto prima di passare alla prova vera e propria. Inoltre considereremo tutte le interpretazioni semantiche depurate dai controlli d'errore: non è difficile infatti vedere che ogni volta che si verifica un errore il sistema di interpretazione semantica delle dispense non fa altro che propagare l'errore, garantendo quindi convergenza. Infine si assume che lo stato σ_k sia equivalente allo stato in forma estesa $< i_k, o_k, m_k >$, quindi ad esempio se scriviamo m_2 sappiamo che questo indica la funzione memoria dello stato σ_2, senza bisogno di appesantire la dimostrazione.

Tutte le interpretazioni verranno calcolate già in un generico stato σ, per semplicità di scrittura.

Iniziamo col definire l'interpretazione delle tre costanti naturali presenti nel comando, ovvero 0, 1 e 3:

$$\mathbb{E}[\![0]\!]\sigma =< 0, \sigma >$$
$$\mathbb{E}[\![1]\!]\sigma =< 1, \sigma >$$
$$\mathbb{E}[\![3]\!]\sigma =< 3, \sigma >$$

Sfruttando la definizione di interpretazione di variabile (depurata da controlli d'errore) e la notazione assunta sugli stati, abbiamo la seguente interpretazione per x in uno stato σ:

$$\mathbb{E}[\![x]\!]\sigma =< mx, \sigma >$$

Vediamo adesso l'interpretazione di $x > 0$, sostituendo opportunamente le interpretazioni già calcolate:

$$\begin{aligned}\mathbb{E}[\![x > 0]\!]\sigma = &\text{ let } < v_1, \sigma_1 > \text{ be } \mathbb{E}[\![x]\!]\sigma \text{ in}\\ &\text{let } < v_2, \sigma_2 > \text{ be } \mathbb{E}[\![0]\!]\sigma_1 \text{ in}\\ &< v_1 \text{ gt } v_2, \sigma_2 >\\ = &<< mx, \sigma > \text{ gt } < 0, \sigma >, \sigma >\end{aligned}$$

Il simbolo *gt* (greater than) è il simbolo semantico corrispondente al simbolo sintattico $-$, nonnché un'istanza di *nop*. Si noti come, grazie al fatto che nessuna delle due valutazioni modifica lo stato, possiamo utilizzare lo stesso σ, alleggerendo così la notazione.

Procediamo in modo analogo per $x - 1$ e per $x < 3$, altre due operazioni tra naturali:

$$\begin{aligned}\mathbb{E}[\![x - 1]\!]\sigma = &\text{ let } < v_1, \sigma_1 > \text{ be } \mathbb{E}[\![x]\!]\sigma \text{ in}\\ &\text{let } < v_2, \sigma_2 > \text{ be } \mathbb{E}[\![1]\!]\sigma_1 \text{ in}\\ &< v_1 \text{ minus } v_2, \sigma_2 >\\ = &<< mx, \sigma > \text{ minus } < 1, \sigma >, \sigma >\end{aligned}$$

$$\begin{aligned}\mathbb{E}[\![x < 3]\!]\sigma &= \text{let} < v_1, \sigma_1 > \text{be } \mathbb{E}[\![x]\!]\sigma \text{ in} \\ &\quad \text{let} < v_2, \sigma_2 > \text{be } \mathbb{E}[\![3]\!]\sigma_1 \text{ in} \\ &\quad < v_1 \text{ lt } v_2, \sigma_2 > \\ &= << mx, \sigma > \text{ lt } < 3, \sigma >, \sigma >\end{aligned}$$

Permettendoci un piccolo abuso di notazione, definiamo le tre notazioni seguenti, relative a due valori naturali e ad uno stato, delle operazioni "maggiore di", "meno" e "minore di", che intuitivamente rappresentano il solo valore ottenuto dall'esecuzione di quelle operazioni su quei valori e su quello stato:

$$\begin{aligned}GT_{i,j,\sigma} &\equiv < i, \sigma > \text{ gt } < j, \sigma > \\ M_{i,j,\sigma} &\equiv < i, \sigma > \text{ minus } < j, \sigma > \\ LT_{i,j,\sigma} &\equiv < i, \sigma > \text{ lt } < j, \sigma >\end{aligned}$$

Inoltre useremo la notazione, in forma compatta, $\sigma[v/x] \equiv < i, o, m[v/x] >$.

Calcoliamo adesso l'interpretazione del comando di assegnazione $x := x - 1$ nello stato σ:

$$\begin{aligned}\mathbb{C}[\![x := x - 1]\!]\sigma &= \text{let} < v_1, \sigma_1 > \text{be } \mathbb{E}[\![x - 1]\!] \text{ in} \\ &\quad < i_1, o_1, m_1[v_1/x] > \\ &= < i, o, m[M_{mx,1,\sigma}/x] > \\ &= \sigma[M_{mx,1,\sigma}/x]\end{aligned}$$

Si può facilmente dimostrare che:

$$\begin{aligned}GT_{mx,j,\sigma[M_{mx,h,\sigma}/x]} &= GT_{mx,(j+h),\sigma} \\ M_{mx,j,\sigma[M_{mx,h,\sigma}/x]} &= M_{mx,(j+h),\sigma} \\ LT_{mx,j,\sigma[M_{mx,h,\sigma}/x]} &= LT_{mx,(j+h),\sigma} \\ \sigma[M_{mx,a,\sigma}/x][M_{mx,b,\sigma[M_{mx,a,\sigma}/x]}/x] &= \sigma[M_{mx,(a+b),\sigma}/x]\end{aligned}$$

Inoltre risulta intuitivo che:

$$\begin{aligned}GT_{a,b,\sigma} \to (GT_{a,c,\sigma} \to \alpha, \gamma), \beta &= GT_{a,b,\sigma} \to \alpha, \beta, && \text{con } b \geq c \\ LT_{a,b,\sigma} \to (GT_{a,c,\sigma} \to \alpha, \gamma), \beta &= LT_{a,b,\sigma} \to \gamma, \beta, && \text{con } b > c + 1 \\ LT_{a,b,\sigma} \to \alpha, (GT_{a,c,\sigma} \to \beta, \gamma) &= LT_{a,b,\sigma} \to \alpha, \beta, && \text{con } b > c \\ LT_{a,b,\sigma} \to (LT_{a,c,\sigma} \to \alpha, \gamma), \beta &= LT_{a,b,\sigma} \to \alpha, \beta, && \text{con } b \leq c\end{aligned}$$

Le proprietà esposte sopra valgono anche quando il comando if interno fa parte di espressioni o comandi più complessi.

Banalmente il comando **noaction** viene interpretato come:

$$\mathbb{C}[\![\textbf{noaction}]\!]\sigma = \sigma$$

Definiamo adesso l'intero comando if presente, $IF \equiv \textbf{if } x < 3 \textbf{ then } x := x-1 \textbf{ else noaction}$. La sua interpretazione semantica sarà quindi:

$$\begin{aligned}\mathbb{C}[\![IF]\!]\sigma &= \text{let} < v_1, \sigma_1 > \text{be } \mathbb{E}[\![x < 3]\!]\sigma \text{ in} \\ &\quad v_1 \to \mathbb{C}[\![x := x - 1]\!]\sigma_1, \mathbb{C}[\![\textbf{noaction}]\!]\sigma_1 \\ &= LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma\end{aligned}$$

Definiamo ora il funzionale della specifica ricorsiva del comando **while** in esame, depurato da controlli d'errore, in modo simile a quanto fatto nelle dispense a pag. 187.:

$$F \equiv \lambda\Theta.\lambda\sigma.\ \text{let} < v_1, \sigma_1 > \text{be } \mathbb{E}[\![x > 0]\!]\sigma \text{ in } v_1 \to \Theta(\mathbb{C}[\![IF]\!]\sigma_1), \sigma_1$$

Sostituendo le interpretazioni semantiche già calcolate otteniamo:

$$F = \lambda\Theta.\lambda\sigma.GT_{mx,0,\sigma} \to \Theta(LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma), \sigma$$

Per definizione, sappiamo che l'interpretazione del comando **while** è data da:

$$\mathbb{C}[\![\textbf{while } x > 0 \textbf{ do if } x < 3 \textbf{ then } x := x - 1 \textbf{ else noaction}]\!] = fix(F)$$

Calcoliamo allora il punto fisso del funzionale F per approssimazioni successive:

$$\begin{aligned}F^0\Omega &= \Omega \\ &= \lambda x.\bot\end{aligned}$$

La prima approssimazione è la meno definita, infatti ci sta dicendo che il comando diverge sempre. Tuttavia noi stiamo cercando un punto fisso, quindi andiamo avanti:

$$\begin{aligned}F^1\Omega &= F(F^0\Omega) \\ &= F\Omega \\ &= \lambda\sigma.GT_{mx,0,\sigma} \to \Omega(LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma), \sigma \\ &= \lambda\sigma.GT_{mx,0,\sigma} \to \bot, \sigma\end{aligned}$$

Il comando è già più definito, in quanto adesso sappiamo che non diverge se il valore assegnato ad x non è maggiore di 0 (comprendendo anche i vari errori che non abbiamo incluso, quali valori booleani assegnati alla x o *unbound*). Vediamo come si comporta la seconda approssimazione:

$$\begin{aligned}F^2\Omega &= F(F^1\Omega) \\ &= \lambda\sigma.GT_{mx,0,\sigma} \to (\lambda\sigma_1.GT_{mx,0,\sigma_1} \to \bot, \sigma_1)(LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma), \sigma \\ &= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \bot, \sigma[M_{mx,1,\sigma}/x]), (GT_{mx,0,\sigma} \to \bot, \sigma)), \sigma \\ &= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \bot, \sigma[M_{mx,1,\sigma}/x]), \bot), \sigma\end{aligned}$$

L'interpretazione del comando **while** inizia a prendere forma: l'approssimazione appena calcolata ci dice che se il valore associato ad x è 2, allora decrementa il suo valore in memoria, altrimenti o diverge o non fa niente. Calcoliamo la prossima approssimazione:

$$
\begin{aligned}
F^3\Omega &= F(F^2\Omega) \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (\lambda\sigma_1.GT_{mx,0,\sigma_1} \to (LT_{mx,3,\sigma_1} \to (GT_{mx,1,\sigma_1} \to \bot, \sigma_1[M_{mx,1,\sigma_1}/x]), \bot), \sigma_1) \\
&\quad (LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma), \sigma \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,0,\sigma[M_{mx,1,\sigma}/x]} \to (LT_{mx,3,\sigma[M_{mx,1,\sigma}/x]} \to \\
&\quad (GT_{mx,1,\sigma[M_{mx,1,\sigma}/x]} \to \bot, \sigma[M_{mx,1,\sigma}/x][M_{mx,1,\sigma[M_{mx,1,\sigma}/x]}/x]), \bot), \sigma[M_{mx,1,\sigma}/x]), \\
&\quad (GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \bot, \sigma[M_{mx,1,\sigma}/x]), \bot), \sigma)), \sigma \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \sigma[M_{mx,2,\sigma}/x], \sigma[M_{mx,1,\sigma}/x]), \bot), \sigma
\end{aligned}
$$

Questa terza approssimazione è molto interessante! Quello che ci dice è che per 0 il comando non fa niente, per valori maggiori o uguali a 3 diverge senza modificare alcun valore, altrimenti decrementa x di 2 o di 1 a seconda che il valore di x sia 2 o 1, rispettivamente. Intuitivamente, questa è proprio la descrizione che ci viene in mente quando pensiamo all'interpretazione di questo comando **while**. Proviamo a vedere se riusciamo a fare di meglio con un'altra approssimazione, ovvero se riusciamo a rendere l'interpretazione del comando ancora più definita:

$$
\begin{aligned}
F^4\Omega &= F(F^3\Omega) \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (\lambda\sigma_1.GT_{mx,0,\sigma_1} \to (LT_{mx,3,\sigma_1} \to (GT_{mx,1,\sigma_1} \to \sigma_1[M_{mx,2,\sigma_1}/x], \\
&\quad \sigma_1[M_{mx,1,\sigma_1}/x]), \bot), \sigma_1)(LT_{mx,3,\sigma} \to \sigma[M_{mx,1,\sigma}/x], \sigma), \sigma \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to (LT_{mx,4,\sigma} \to \\
&\quad (GT_{mx,2,\sigma} \to \sigma[M_{mx,3,\sigma}/x], \sigma[M_{mx,2,\sigma}/x]), \bot), \sigma[M_{mx,1,\sigma}/x]), (GT_{mx,0,\sigma} \to \\
&\quad (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \sigma[M_{mx,2,\sigma}/x], \sigma[M_{mx,1,\sigma}/x]), \bot), \sigma)), \sigma \\
&= \lambda\sigma.GT_{mx,0,\sigma} \to (LT_{mx,3,\sigma} \to (GT_{mx,1,\sigma} \to \sigma[M_{mx,2,\sigma}/x], \sigma[M_{mx,1,\sigma}/x]), \bot), \sigma \\
&= F^3\Omega
\end{aligned}
$$

Abbiamo trovato un punto fisso! La terza approssimazione calcolata prima risulta essere un punto fisso del funzionale F (il minimo, grazie al Teorema di Kleene) in quanto $F(F^3\Omega) = F^3\Omega$. Questo ci dice che la descrizione informale data prima del comando è *esattamente* l'interpretazione del comando e non una semplice approssimazione.

Quindi, ricapitolando, possiamo dedurre che l'insieme degli stati che fanno divergere il comando proposto sono tutti quei stati che fanno finire l'interpretazione semantica del **while** nel ramo contenente $\bot$, ovvero tutti quegli stati con input e output qualsiasi e con valori in memoria associati alla variabile x maggiori o uguali a 3. Come già detto all'inizio, è facile vedere come i valori per x che causano errore, quali valori booleani o *unbound*, garantiscano la convergenza, in quanto generano un errore che viene subito propagato, bloccando l'esecuzione del programma.

Esercizio 9.6

Si aggiunga al linguaggio SMALL un comando **stop** con la semantica informale di far terminare il programma. Se ne dia la semantica e si dimostri che $c_1; \mathbf{stop}$ e $c_1; \mathbf{stop}; c_2$ sono semanticamente equivalenti.

Banalmente, essendo **stop** un nuovo comando, per aggiornare la semantica di SMALL basta aggiungere una nuova clausola per i comandi contenente **stop**, quindi si avrà la sintassi dei comandi come segue:

$$
\begin{aligned}
c \quad ::= \quad & e := e_1 \mid c_1; c_2 \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid \mathbf{output}\ e \\
& \mid \mathbf{begin}\ d; c\ \mathbf{end} \mid e(e_1) \mid \mathbf{stop}
\end{aligned}
$$

Ci possono essere varie soluzioni per definire la semantica del nuovo comando **stop**. La più semplice è quella di far produrre al comando **stop** un nuovo elemento, che chiameremo *stop*, e trattarlo im modo simile all'elemento *error*. Infatti se ci pensiamo bene, un errore fa quello che vorrebbe fare il comando **stop**, ovvero terminare il programma senza eseguire altri comandi. L'unica differenza è che, mentre *error* propaga l'errore anche come output del programma stesso, utilizzando **stop** ci aspettiamo invece di fornire comunque un qualche tipo di output, in particolare il contenuto della memoria nella locazione riservata *lout*.

Per far questo, ridefiniamo allora l'operatore $\star$ introdotto nelle dispense. In particolare, definiamo una variante di $\star$, che chiameremo $\overline{\star}$, e che è definito come segue:

Se $f : D_1 \to (D_2 + \{error\} + \{< stop, \sigma >\})$ e $g : D_2 \to (D_3 + \{error\} + \{< stop, \sigma >\})$, allora

$$f \,\overline{\star}\, g : D_1 \to (D_3 + \{error\} + \{< stop, \sigma >\})$$

è definita come segue:

$$\begin{aligned} f \,\overline{\star}\, g = \lambda x.\ & cases\ fx\ of \\ & d : gd; \\ & error : error; \\ & < stop, \sigma > : < stop, \sigma > \\ & endcases \end{aligned}$$

Si intuisce già che l'idea è quella di, non appena si incontra il comando **stop**, propagare una coppia composta dall'elemento *stop* e da una memoria fino all'interpretazione del programma stesso, che restituirà quindi l'output in tale memoria.

La semantica dei programmi diventa quindi:

$$\begin{aligned} \mathcal{P}[\![\textbf{program}\ c]\!]in = & cases\ (\mathcal{C}[\![c]\!]\rho_0(\lambda x.unused)[in/lin][nil/lout])\ of \\ & \sigma : \sigma\ lout; \\ & error : error; \\ & < stop, \sigma > : \sigma\ lout \\ & endcases \end{aligned}$$

L'interpretazione del comando **stop** è quindi definita come segue:

$$\mathcal{C}[\![\textbf{stop}]\!]\rho = \lambda\sigma. < stop, \sigma >$$

Risulta chiaro quindi che è necessario modificare il dominio semantico della funzione d'interpretazione, aggiungendo il caso in cui l'interpretazione di un comando produce la coppia $< stop, \sigma >$, ovvero:

$$\mathcal{C} : Com \to \mathbb{AMB} \to \mathbb{MEM} \to (\mathbb{MEM} + \{error\} + (\{stop\} \times \mathbb{MEM}))$$

In aggiunta a queste modifiche, per rendere il nuovo meccanismo funzionante, c'è bisogno di aggiungere la funzionalità di propagare *stop* a tutto il sistema di interpretazione. Per fare ciò basterà sostituire l'operarore $\overline{\star}$ all'operatore $\star$ all'interno delle funzioni di interpretazione semantica dei comandi già visti. Tutti gli altri rimarranno invece il vecchio operatore $\star$ già definito. Non importa, infatti, andare a modificare l'interpretazione di espressioni e dichiarazioni: per quanto riguarda le espressioni, vediamo che esse non contengono in alcun modo comandi,

quindi tantomeno **stop**, e quindi il loro comportamento rimane inalterato; si potrebbe pensare che le dichiarazioni diano problemi, in quanto si utilizza la funzione d'interpretazione dei comandi per definire l'interpretazione della dichiarazione di procedure, tuttavia nemmeno questo caso da problemi in quanto, se anche una procedura contenesse **stop** al suo interno, ogni volta che si invoca una procedura, il corpo della procedura viene sostituito all'invocazione stessa (con il parametro attuale sostituito al parametro formale) e quindi si "trasforma" in un semplice comando all'interno del blocco in cui è avvenuta l'invocazione, riconducendo il tutto a casi già noti.

Per concludere l'esercizio, mostriamo che i comandi $c_1;\mathbf{stop}$ e $c_1;\mathbf{stop};c_2$ risultano semanticamente equivalenti secondo la semantica appena descritta. Si distinguono tre casi, a seconda che il comando c_1 contenga **stop** o meno e che generi un errore o meno.

- c_1 non contiene **stop** e produce la memoria σ_1:

$$\begin{aligned}\mathcal{C}[\![c_1;\mathbf{stop}]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= \sigma_1 \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= (\lambda\sigma'.(\lambda\sigma''.< stop,\sigma'' >)\sigma')\sigma_1 \qquad \text{il valore viene fatto passare da } \overline{\star} \\ &= < stop,\sigma_1 >\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![c_1;\mathbf{stop};c_2]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= \sigma_1 \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= (\lambda\sigma'.(\lambda\sigma''.< stop,\sigma'' >)\sigma')\sigma_1 \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= < stop,\sigma_1 > \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= < stop,\sigma_1 > \qquad \text{lo } stop \text{ viene propagato da } \overline{\star}\end{aligned}$$

  Sfruttando le proprietà del nuovo operatore, abbiamo dimostrato che, dati un ambiente ed una memoria qualsiasi, i due comandi producono la stessa coppia, ovvero sono semanticamente equivalenti.
  In modo analogo vediamo il caso in cui c_1 contenga **stop**.

- c_1 contiene **stop** e produce la coppia $< stop,\sigma_1 >$:

$$\begin{aligned}\mathcal{C}[\![c_1;\mathbf{stop}]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= < stop,\sigma_1 > \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= < stop,\sigma_1 > \qquad \text{lo } stop \text{ viene propagato da } \overline{\star}\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![c_1;\mathbf{stop};c_2]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= < stop,\sigma_1 > \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\ &= < stop,\sigma_1 > \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \qquad \text{lo } stop \text{ viene propagato da } \overline{\star} \\ &= < stop,\sigma_1 > \qquad \text{lo } stop \text{ viene propagato da } \overline{\star}\end{aligned}$$

  Anche in questo caso risultano semanticamente equivalenti.
  Rimane da far vedere che in caso d'errore, l'operatore $\overline{\star}$ si comporta esattamente come $\star$.

- c_1 produce *error*:

$$\begin{aligned}\mathcal{C}[\![c_1;\mathbf{stop}]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= error \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \\ &= error \qquad \textit{error} \text{ viene propagato da } \overline{\star}\end{aligned}$$

$$
\begin{aligned}
\mathcal{C}[\![c_1; \mathbf{stop}; c_2]\!]\rho\sigma &= \mathcal{C}[\![c_1]\!]\rho\sigma \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\
&= error \mathbin{\overline{\star}} (\lambda\sigma'.\mathcal{C}[\![\mathbf{stop}]\!]\rho\sigma') \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) \\
&= error \mathbin{\overline{\star}} (\lambda\overline{\sigma}.\mathcal{C}[\![c_2]\!]\rho\overline{\sigma}) && error \text{ viene propagato da } \overline{\star} \\
&= error && error \text{ viene propagato da } \overline{\star}
\end{aligned}
$$

Abbiamo quindi dimostrato che i due comandi proposti, sotto la semantica descritta, risultano equivalenti, ovvero $c_1; \mathbf{stop} \equiv c_1; \mathbf{stop}; c_2$.

Esercizio 11.3

Esercizio 11.5

Esercizio 12.2