Elementi di Semantica Operazionale

Appunti per gli studenti di Fondamenti di Programmazione

Corso di Laurea in Informatica Applicata
Polo Universitario G. Marconi - La Spezia

Università di Pisa
A.A. 2006/07

R. Barbuti, P. Mancarella, M.E.Occhiuto e F. Turini

Indice

Premessa

Gli studenti sono invitati a leggere queste note con criticità ed attenzione, per la potenziale presenza di refusi ed imprecisioni che possono essere rimasti nel testo. Si ringraziano tutti coloro che vorranno segnalarli; commenti, suggerimenti e critiche riguardanti il materiale o la sua presentazione sono particolarmente benvenuti.

Il materiale di queste note nasce dallo sforzo congiunto di docenti, assistenti e studenti dei corsi di Teoria ed Applicazione delle Macchine Calcolatrici (TAMC) e di Programmazione I dei Corsi di Laurea in Scienze dell'Informazione e in Informatica, per aggiornare e migliorare il corso. Per il loro contributo in tal senso, nonché per le molte osservazioni sul testo, ringraziamo Antonio Brogi, Paola Cappanera, Stefano Cataudella, Andrea Corradini, Gianluigi Ferrari, Paola Inverardi, Carlo Montangero, Monica Nesi, Dino Pedreschi, Alessandra Raffaetà e gli studenti dei corsi di TAMC, Programmazione I e Fondamenti di Programmazione degli Anni Accademici precedenti.

1 Introduzione

I linguaggi di programmazione sono linguaggi artificiali che servono per esprimere algoritmi, ovvero procedimenti risolutivi di problemi, in modo che un elaboratore automatico sia poi in grado di eseguirli. Un *programma* non è altro che la traduzione di un algoritmo in un particolare linguaggio di programmazione.

La definizione formale dei linguaggi, artificiali e non, prevede essenzialmente due aspetti: una descrizione della *sintassi* del linguaggio, ovvero delle frasi legali esprimibili nel linguaggio stesso; e una descrizione della *semantica* del linguaggio, ovvero del significato di frasi sintatticamente corrette. Nel caso dei linguaggi di programmazione, la sintassi descrive la struttura di un programma, ovvero in che modo frasi e simboli di base possono essere composti al fine di ottenere un programma legale nel linguaggio. A differenza dei linguaggi naturali, la sintassi dei linguaggi di programmazione è relativamente semplice e può essere descritta in modo rigoroso utilizzando i formalismi messi a disposizione dalla teoria dei linguaggi formali. Non è scopo di queste note, né del corso di Fondamenti di Programmazione, affrontare lo studio di tale teoria: nel seguito daremo per scontati i pochi concetti di base sulla sintassi dei linguaggi introdotti nella prima parte del corso.

Nella storia, peraltro recente, dei linguaggi di programmazione vi è un evidente contrasto tra il modo in cui ne viene descritta la sintassi e quello in cui ne viene descritta la semantica. Di solito una descrizione rigorosa e formale della sintassi è affiancata da una descrizione poco rigorosa e del tutto informale della semantica: i manuali di riferimento dei linguaggi di programmazione più comuni testimoniano questa situazione. Le descrizioni informali sono spesso incomplete e poco precise, dando luogo ad incomprensioni da parte di chi ne fa uso, vuoi per capire come utilizzare un dato linguaggio nella risoluzione di problemi, vuoi per affrontare il problema della realizzazione (o implementazione) di un dato linguaggio, vuoi per ragionare in modo astratto sulla correttezza dei programmi scritti in un dato linguaggio.

A tali carenze si è cercato di sopperire introducendo metodi e tecniche formali per la definizione della semantica dei linguaggi di programmazione. Scopo di queste note è introdurre il lettore ad una di esse che va sotto il nome di *semantica operazionale*. L'idea che sta alla base di tale approccio è di dare la semantica di un linguaggio mediante la definizione di un *sistema* e del suo comportamento in corrispondenza di programmi scritti nel linguaggio.

Una prima importante osservazione da fare è che nella descrizione della semantica di un linguaggio si fa riferimento alla sua *sintassi astratta*, intendendo con essa una descrizione sintattica che pone in rilievo la struttura sintattica essenziale dei vari costrutti del linguaggio, prescindendo da dettagli che non contribuiscono a chiarire il significato delle frasi, anche se rilevanti sotto altri punti di vista. Un modo per descrivere la sintassi astratta dei vari costrutti è attraverso i cosiddetti *alberi di derivazione*, in cui è specificata proprio la struttura, sotto forma di albero, dei costrutti stessi. Tuttavia, per non appesantire la notazione, presenteremo sia le produzioni della grammatica utilizzata sia i vari costrutti sintattici sotto forma di stringhe di simboli, ma dovrà essere sempre chiaro al lettore che esse rappresentano in realtà l'albero di derivazione per la stringa stessa.

2 I sistemi di transizioni

In questo paragrafo introduciamo i concetti che sono alla base dell'approccio *operazionale* alla semantica dei linguaggi e diamo la semantica in questo stile di un semplice linguaggio per la descrizione di espressioni.

Definizione 2.1 Un sistema di transizioni $\mathtt{S}$ è una tripla $\langle \Gamma, \mathtt{T}, \rightarrow \rangle$ dove

- Γ è un insieme i cui elementi sono detti *configurazioni*;
- $\mathtt{T} \subseteq \Gamma$ è un sottoinsieme di Γ i cui elementi sono detti *configurazioni terminali*;
- $\rightarrow$ è un insieme di coppie $\langle \gamma, \gamma' \rangle$ di configurazioni e viene detta *relazione di transizione*. Denoteremo con $\gamma \rightarrow \gamma'$ l'appartenenza della coppia $\langle \gamma, \gamma' \rangle$ alla relazione $\rightarrow$ e chiameremo *transizioni* tali elementi. □

In prima approssimazione possiamo pensare alle configurazioni come agli *stati* in cui il sistema si può trovare durante il suo funzionamento: in particolare le configurazioni terminali corrispondono a quegli stati in cui il sistema ha terminato le sue operazioni. La relazione di transizione descrive il comportamento del sistema: il significato intuitivo di una transizione $\gamma \to \gamma'$ è che il sistema dallo stato γ è in grado di portarsi nello stato γ'. In quest'ottica, il comportamento del sistema a partire da una certa configurazione è formalizzato nella seguente definizione.

Definizione 2.2 Dato un sistema di transizioni $\mathtt{S} = \langle \Gamma, \mathtt{T}, \to \rangle$, una *derivazione* in $\mathtt{S}$ è una sequenza (eventualmente infinita) di configurazioni $\gamma_0, \gamma_1, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ tale che, per ogni $k \geq 1$, $\gamma_{k-1} \to \gamma_k$. Nel seguito, una derivazione in $\mathtt{S}$ verrà indicata con $\gamma_0 \to \gamma_1 \to \ldots \to \gamma_i$. Inoltre indicheremo con $\gamma \xrightarrow{*} \gamma'$ l'esistenza di una derivazione $\gamma_0 \to \gamma_1 \to \ldots \to \gamma_i$ in cui $\gamma_0 = \gamma$ e $\gamma_i = \gamma'$. □

Dato un sistema di transizioni $\langle \Gamma, \mathtt{T}, \to \rangle$, diremo che una configurazione $\gamma \in \Gamma \setminus \mathtt{T}$ è *bloccata* se e solo se non esiste alcuna configurazione γ' tale che $\gamma \to \gamma'$.

Come primi esempi, consideriamo le grammatiche e costruiamo dei sistemi di transizioni ad essi corrispondenti.

Esempio 2.3 Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera da contesto. Definiamo un sistema di transizioni S_G associato alla grammatica nel seguente modo.

- le configurazioni di S_G sono stringhe in $(\Lambda \cup V)^*$;
- le configurazioni terminali di S_G sono le stringhe in Λ^*;
- dette α e β due configurazioni (stringhe), esiste una transizione $\alpha \to_G \beta$ se e solo se
  - α è del tipo $\delta A \gamma$, con $\delta, \gamma \in (\Lambda \cup V)^*$ e $A \in V$;
  - β è del tipo $\delta \eta \gamma$;
  - $A ::= \eta$ è una produzione di P.

È evidente, per costruzione di S_G, che $\alpha \in \mathcal{L}(G)$ se e solo se $S \xrightarrow{*}_G \alpha$, dove $\mathcal{L}(G)$ indica il linguaggio generato da G. ■

2.1 Regole condizionali

Nel seguito utilizzeremo sistemi di transizioni in cui la relazione di transizione è spesso espressa mediante regole *condizionali*, ovvero regole del tipo:

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \to \gamma'}$$

dove $\pi_1, \pi_2 \ldots \pi_n$ (dette *precondizioni* o *premesse* della regola) sono i prerequisiti alla transizione $\gamma \to \gamma'$. Così, ad esempio, la definizione di $\to_G$ dell'esempio 2.3 può essere data come segue. Per ogni produzione $(A ::= \eta) \in P$ si definisce una regola condizionale:

$$\frac{\alpha = \delta A \gamma \qquad \delta, \gamma \in (\Lambda \cup V)^* \qquad \beta = \delta \eta \gamma}{\alpha \to_G \beta}$$

Se consideriamo, ad esempio, la grammatica definita dalle seguenti produzioni:

```
S ::= ab | aSb
```

che genera il linguaggio $\{\mathtt{a}^n\mathtt{b}^n \mid n > 0\}$. Le regole condizionali del corrispondente sistema di transizioni sono:

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \rightarrow \beta}$$

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{aSb} \gamma}{\alpha \rightarrow \beta}$$

Più precisamente, quando scriviamo

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \rightarrow \gamma'}$$

stiamo in realtà esprimendo un insieme (in generale infinito) di regole, come risulterà chiaro più avanti in questa sezione [1].

Inoltre è necessario essere precisi su quale forma possano assumere le premesse delle regole condizionali, al fine di evitare di scrivere regole prive di significato operazionale. Riconsideriamo, a questo scopo, la regola data in precedenza:

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \rightarrow \beta}$$

Si noti che sia nella conclusione che nelle premesse della regola compaiono:

- nomi, come $\alpha, \beta, \gamma, \delta$ che chiameremo *variabili* e che rappresentano generici elementi di un opportuno dominio del discorso (ad esempio, nel caso specifico, α, β, γ e δ rappresentano stringhe);
- costanti, come $\mathtt{S}, \mathtt{a}, \mathtt{b}$ che stanno per oggetti specifici e noti (ad esempio, $\mathtt{S}$ è il simbolo iniziale della grammatica in esame, $\mathtt{a}$ e $\mathtt{b}$ sono i simboli nell'alfabeto della grammatica).

Le premesse della regola sono poi *predicati* o *relazioni* tra espressioni (termini) costruite a partire dalle variabili, dalle costanti e da operatori noti. Ad esempio, la premessa $\alpha = \delta \mathtt{S} \gamma$ esprime un'uguaglianza tra stringhe in cui compaiono:

- le variabili α, γ e δ,
- la costante $\mathtt{S}$,
- l'operatore $=$ di uguaglianza tra stringhe e l'operatore di concatenazione tra stringhe nell'espressione $\delta \mathtt{S} \gamma$(si noti che tale operatore è rappresentato dalla semplice giustapposizione delle stringhe in questione).

Lo scopo di tale premessa è di specificare le parti che compongono la stringa α: un generico prefisso, chiamato δ, seguito dalla categoria sintattica $\mathtt{S}$, seguita dalla parte conclusiva, chiamata γ. Si noti che tali variabili vengono utilizzate anche nella premessa $\beta = \delta \mathtt{ab} \gamma$, esprimendo così il fatto che le stringhe α e β hanno in comune il prefisso (δ) e la parte conclusiva (γ).

Ancora, la premessa $\delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^*$ esprime il fatto che le componenti δ e γ delle stringhe α e β rappresentano stringhe costruite a partire da simboli terminali e categorie sintattiche della grammatica in esame: in tale premessa vengono utilizzati gli operatori noti $\cup$, * e la relazione di appartenenza $\in$.

Come vedremo nel seguito, in molti casi consentiremo anche che le premesse di una regola che definiscono una relazione di transizione $\rightarrow$ possano contenere istanze di transizioni del tipo $\delta \rightarrow' \delta'$ in cui $\rightarrow'$ può essere sia la relazione di transizione di un altro sistema di transizioni sia la relazione di transizione $\rightarrow$ stessa del sistema in esame. In questo secondo caso parleremo di regola *ricorsiva*.

Riassumiamo quanto detto circa le regole condizionali che definiscono una relazione di transizione. Data una regola del tipo:

[1] Dovremmo quindi parlare, più che di regola, di *schema di regola*.

$$\boxed{\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \to \gamma'} \qquad (R)}$$

ciascuna premessa può essere:

(i) un'uguaglianza del tipo $x = t$, dove x è una variabile e t è una espressione (termine) costruita a partire da costanti, variabili e operatori elementari;

(ii) un predicato del tipo $t \; rel \; t'$, dove t, t' sono termini e rel è un operatore di relazione elementare;

(iii) una transizione del tipo $\delta \to \delta'$, dove δ, δ' sono configurazioni e $\to$ è la relazione di transizione di un sistema di transizioni (che può anche essere il sistema che si sta definendo).

È importante osservare che le variabili utilizzate in una regola condizionale devono essere introdotte allo scopo di determinare condizioni significative per la applicabilità della regola stessa. Nell'esempio precedente, le variabili δ e γ sono state introdotte al fine di mettere in relazione la struttura delle stringhe α e β presenti nella conclusione della regola stessa, come abbiamo già messo in evidenza. Bisogna fare attenzione a non introdurre variabili inutili e condizioni su di esse che potrebbero rendere la regola inapplicabile. Si consideri ad esempio la seguente (pseudo) regola:

$$\frac{\alpha, \beta, \gamma \in \{\mathtt{a}, \mathtt{b}\}^* \quad \alpha = \mathtt{a}\beta\gamma \qquad n, m > 0 \qquad n = m + 1}{\alpha \to \beta}$$

Mentre è chiaro il ruolo giocato dalla variabile γ, come parte conclusiva della stringa α, è invece oscuro quello giocato dalle variabili n, m a valori naturali. Osserviamo che l'attribuzione di valori concreti alle variabili n e m potrebbe rendere inapplicabile la regola, secondo la definizione di istanziazione o applicabilità data più avanti.

La seguente nozione di *copertura* delle variabili presenti in una regola di transizione consente di stabilire se una regola è ben definita per quanto concerne l'uso delle variabili in essa contenute (anche se ciò non garantisce la correttezza della regola stessa).

Intuitivamente, una variabile è coperta se il suo valore, o direttamente o usando le premesse, è ottenibile dalla configurazione a cui la regola risulta applicabile.

Nella definizione, data una configurazione γ (risp. termine t), indichiamo con $Vars(\gamma)$ (risp. $Vars(t)$) l'insieme di tutte e sole le variabili che compaiono in essa. Analogamente, data una regola condizionale R, indichiamo con $Vars(R)$ l'insieme di tutte e sole le variabili che vi compaiono.

Definizione 2.4 (Copertura delle variabili)
Sia R una regola condizionale

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \to \gamma'}$$

e sia $x \in Vars(R)$. Diciamo allora che x è *coperta* in R se e solo se vale una delle seguenti condizioni:

(1) x occorre in γ;

(2) esiste in R una premessa π_i del tipo $y = t$ tale che y è coperta in R e $x \in Vars(t)$;

(3) esiste in R una premessa π_i del tipo $x = t$ tale che tutte le variabili in $Vars(t)$ sono coperte;

(4) esiste in R una premessa π_i del tipo $\delta \to' \delta'$ tale che tutte le variabili in δ sono coperte in R e $x \in Vars(\delta')$ □

È utile osservare che le eventuali premesse del tipo $t \; rel \; t'$, con rel operatore di relazione diverso dall'uguaglianza, non contribuiscono alla copertura delle variabili: tali premesse vengono introdotte infatti allo scopo di verificare l'applicabilità di una regola ed esprimono condizioni che hanno a che fare con le componenti delle configurazioni presenti nella regola stessa.

Vediamo alcuni esempi di regole condizionali e di copertura delle loro variabili.

Esempio 2.5 Consideriamo di nuovo la regola condizionale

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \to \beta}$$

introdotta in precedenza. Le variabili di tale regola sono tutte e sole quelle nell'insieme $\{\alpha, \beta, \gamma, \delta\}$. Osserviamo che:

- α è coperta (caso (1) della Definizione 2.4);
- δ e γ sono coperte (caso (2) della Definizione 2.4 e punto precedente);
- β è coperta (caso (3) della Definizione 2.4 e punto precedente).

Un modo alternativo per esprime la precedente regola è il seguente

$$\frac{\delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^*}{\delta \mathtt{S} \gamma \to \delta \mathtt{ab} \gamma}$$

le cui variabili δ e γ sono evidentemente coperte.

Consideriamo ora la regola

$$\frac{\alpha, \beta, \gamma \in \{\mathtt{a}, \mathtt{b}\}^* \quad \alpha = \mathtt{a} \beta \gamma \qquad n, m > 0 \qquad n = m + 1}{\alpha \to \beta}$$

le cui variabili sono tutte e sole quelle nell'insieme $\{\alpha, \beta, \gamma, n, m\}$. Mentre è chiaro che α, β e γ sono tutte coperte, ciò non vale per le variabili n ed m.

Non diamo, per il momento, esempi di regole condizionali che contengono premesse del tipo $\delta \to' \delta'$: la loro utilità risulterà chiara nel seguito.

Nel resto di queste note considereremo solo regole le cui variabili sono tutte coperte secondo quanto stabilito dalla Definizione 2.4.

In che senso un insieme di regole condizionali definisce la relazione di transizione $\to$ di un sistema di transizioni $\mathtt{S} = \langle \Gamma, \mathtt{T}, \to \rangle$? Per rispondere a questa domanda dobbiamo dare un procedimento che ci consenta di stabilire quando una coppia (γ, γ'), con $\gamma, \gamma' \in \Gamma$, appartiene alla relazione $\to$ secondo quanto stabilito dalle regole condizionali.

Riprendiamo l'esempio riportato all'inizio di questa sezione, in cui la relazione di transizione del sistema è definita dalle seguenti regole condizionali.

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \to \beta} \qquad \text{(r1)}$$

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{aSb} \gamma}{\alpha \to \beta} \qquad \text{(r2)}$$

e consideriamo la configurazione $\mathtt{aaSbb}$. Vogliamo verificare l'esistenza o meno di una configurazione λ tale che $\mathtt{aaSbb} \to \lambda$. Prendiamo in esame la regola (r1), con $\alpha = \mathtt{aaSbb}$ e osserviamo che:

- ponendo $\delta = \mathtt{aa}$ e $\gamma = \mathtt{bb}$, la precondizione $\mathtt{aaSbb} = \delta \mathtt{S} \gamma$ è verificata;
- per costruzione δ e γ soddisfano la proprietà espressa dalla seconda precondizione;

- ponendo $\beta = \delta\texttt{ab}\gamma$, ovvero $\beta = \texttt{aaabbb}$, anche l'ultima precondizione della regola è soddisfatta;
- dalle due considerazioni precedenti, la regola (r1) ci permette di concludere l'esistenza della transizione $\texttt{aaSbb} \rightarrow \texttt{aaabbb}$.

Abbiamo dunque concluso, dalla regola (r1), l'esistenza di una configurazione $\lambda = \texttt{aaabbb}$ tale che $\texttt{aaSbb} \rightarrow \lambda$.

Il procedimento che abbiamo intrapreso nell'esempio consiste nel determinare dei valori concreti da utilizzare per rimpiazzare le variabili presenti in una regola, in modo da ottenere un caso particolare della regola stessa, detta *istanza* della regola, in cui non occorrono più variabili e tutte le precondizioni sono soddisfatte. Chiameremo questo procedimento *istanziazione* di una regola.

Si noti che, nello stesso esempio, avremmo potuto arrivare alla conclusione che esiste la transizione $\texttt{aaSbb} \rightarrow \texttt{aaaSbbb}$ applicando un procedimento analogo ma a partire dalla regola (r2)[2].

Consideriamo ora la configurazione $\texttt{aabb}$. Osserviamo che non è possibile *istanziare* opportunamente né la regola (r1) né la regola (r2) in modo che le rispettive premesse siano soddisfatte. In particolare, in entrambi i casi non è possibile determinare valori concreti per le variabili δ e γ in modo che $\texttt{aabb} = \delta\texttt{S}\gamma$. Possiamo dunque dire che, la relazione di transizione definita dalle regole (r1) e (r2) non contiene alcuna coppia $(\texttt{aabb}, \lambda)$.

Per formalizzare il processo di istanziazione di una regola introduciamo il concetto di *sostituzione*.

Definizione 2.6 (Sostituzione)
Sia $V = \{x_1, \ldots, x_n\}$ un insieme di variabili distinte e sia $c_1, \ldots, c_n$ una sequenza di costanti. L'insieme di associazioni $\vartheta = \{c_1/x_1, \ldots, c_n/x_n\}$ è detta *sostituzione* per le variabili in V. □

Una volta definita una sostituzione, essa può essere applicata per *istanziare* oggetti in cui compaiono le variabili coinvolte nella sostituzione: nel nostro caso specifico applicheremo sostituzioni alle espressioni (termini) o alle configurazioni presenti in una regola condizionale.

Definizione 2.7 (Applicazione di una sostituzione)
Sia X un termine o una configurazione e sia $V = Vars(X)$ l'insieme delle variabili che vi compaiono. Sia inoltre ϑ una sostituzione per le variabili in V. L'applicazione di ϑ a X, denotata da $(X)\vartheta$ è il termine o la configurazione che si ottiene rimpiazzando in X ogni variabile x con la costante c tale che $c/x \in \vartheta$. Il termine o configurazione $(X)\vartheta$ verrà detto *istanza di X via ϑ*. □

Ad esempio, data la configurazione $\delta S\gamma$ dell'esempio precedente e la sostituzione $\vartheta = \{\texttt{aa}/\gamma, \texttt{bb}/\delta\}$, l'istanza $(\delta S\gamma)\vartheta$ è la configurazione $\texttt{aaSbb}$. Come ulteriore esempio, data l'espressione $E = x > y+1$ e la sostituzione $\vartheta = \{3/x, 0/y\}$, l'istanza $(E)\vartheta$ è l'espressione $3 > 0+1$. Data invece $\vartheta' = \{3/x, 5/y\}$, l'istanza $(E)\vartheta'$ è l'espressione $3 > 5 + 1$. Quest'ultimo esempio mette in luce che l'operazione di istanziazione è puramente *sintattica*: istanziazioni diverse della stessa espressione danno luogo ad espressioni che hanno, in generale, *significati* diversi. Ad esempio, utilizzando il significato comune dei simboli che vi compaiono, la relazione $(E)\vartheta$, ovvero $3 > 0 + 1$, è vera mentre è falsa la relazione $(E)\vartheta'$, ovvero $3 > 5 + 1$.

È evidente che il concetto di istanziazione dato nella Definizione 2.7 può essere facilmente esteso al caso di regole condizionali.

Definizione 2.8 (Istanza di una regola)
Sia R una regola condizionale, sia $V = Vars(R)$ l'insieme di tutte e sole le variabili che occorrono in R e sia ϑ una sostituzione per le variabili in V. L'*istanza di R via ϑ* è la regola condizionale che si ottiene applicando ϑ a tutti i termini e a tutte le configurazioni che occorrono in R. □

Consideriamo ad esempio la regola (r2) data in precedenza

$$\frac{\alpha = \delta\texttt{S}\gamma \quad \delta, \gamma \in (\{\texttt{a}, \texttt{b}\} \cup \{\texttt{S}\})^* \quad \beta = \delta\texttt{aSb}\gamma}{\alpha \rightarrow \beta} \qquad \text{(r2)}$$

[2] Dimostrarlo per esercizio.

e la sostituzione $\vartheta = \{\texttt{aaSbb}/\alpha, \texttt{aa}/\delta, \texttt{bb}/\gamma, \texttt{aaaSbbb}/\beta\}$. L'istanza di (r2) via ϑ è la regola condizionale

$$\frac{\texttt{aaSbb} = \texttt{aaSbb} \quad \texttt{aa}, \texttt{bb} \in (\{\texttt{a}, \texttt{b}\} \cup \{\texttt{S}\})^* \quad \texttt{aaaSbbb} = \texttt{aaaSbbb}}{\texttt{aaSbb} \rightarrow \texttt{aaaSbbb}} \quad (\text{r2})\vartheta$$

Osserviamo che le premesse dell'istanza così ottenuta sono tutte soddisfatte. Consideriamo ora la sostituzione $\vartheta' = \{\texttt{aaSbb}/\alpha, \texttt{a}/\delta, \epsilon/\gamma, \texttt{aaSb}/\beta\}$. L'istanza di (r2) via ϑ' è la regola condizionale

$$\frac{\texttt{aaSbb} = \texttt{aS} \quad \texttt{a}, \epsilon \in (\{\texttt{a}, \texttt{b}\} \cup \{\texttt{S}\})^* \quad \texttt{aaSb} = \texttt{aaSb}}{\texttt{aaSbb} \rightarrow \texttt{aaSb}} \quad (\text{r2})\vartheta'$$

Questa volta, nell'istanza ottenuta le premesse non sono tutte soddisfatte. In particolare è falsa la prima premessa $\texttt{aaSbb} = \texttt{aS}$, e ciò ad indicare che la scelta delle costanti associate alle variabili α, δ e γ non consente una corretta "scomposizione" della stringa associata ad α in $\delta\texttt{S}\gamma$.

Siamo finalmente in grado di definire precisamente in che senso un insieme di regole condizionali definisce una relazione di transizione in un sistema di transizioni.

Definizione 2.9 Sia $\texttt{S} = \langle \Gamma, \texttt{T}, \rightarrow \rangle$ un sistema di transizioni la cui relazione di transizione $\rightarrow$ è definita mediante un insieme finito di regole condizionali $(r_1), \ldots, (r_n)$. Date due configurazioni $\gamma, \gamma' \in \Gamma$, la coppia (γ, γ') appartiene alla relazione $\rightarrow$ se e soltanto se esiste una sostituzione ϑ per le variabili in $Vars(r_i)$, con $1 \leq i \leq n$, tale che:

- tutte le premesse di $(r_i)\vartheta$ sono soddisfatte;
- la conclusione di $(r_i)\vartheta$ è $\gamma \rightarrow \gamma'$. □

Consideriamo come esempio le regole di transizione $(r1)$ e $(r2)$ date in precedenza e osserviamo:

- $\texttt{aaSbb} \rightarrow \texttt{aaaSbbb}$: infatti la sostituzione ϑ vista in precedenza produce un'istanza della regola $(r2)$ le cui premesse sono tutte soddisfatte;
- $\texttt{aaSbb} \not\rightarrow \texttt{aaSaSbb}$: è facile convincersi, infatti, che non è possibile individuare una sostituzione ϑ in modo la conclusione di $(r1)\vartheta$ o $(r2)\vartheta$ sia esattamente $\texttt{aaSbb} \rightarrow \texttt{aaSaSbb}$ e in modo che le premesse delle regole così istanziate siano tutte soddisfatte[3].

La Definizione 2.9 suggerisce anche un modo per determinare, a partire da una configurazione γ, una configurazione γ' tale che $\gamma \rightarrow \gamma'$, se quest'ultima esiste. Vediamolo attraverso un esempio, utilizzando ancora le regole $(r1)$ e $(r2)$ viste in precedenza. Supponiamo di voler determinare, se esiste, una configurazione β tale che $\texttt{abaSbb} \rightarrow \beta$. Analizziamo dapprima la regola $(r1)$:

$$\frac{\alpha = \delta\texttt{S}\gamma \quad \delta, \gamma \in (\{\texttt{a}, \texttt{b}\} \cup \{\texttt{S}\})^* \quad \beta = \delta\texttt{ab}\gamma}{\alpha \rightarrow \beta} \quad (\text{r1})$$

È evidente che una qualunque sostituzione che consenta di applicare la Definizione 2.9 deve contenere l'associazione $\texttt{abaSbb}/\alpha$. L'istanziazione (parziale) della premessa

$$\alpha = \delta\texttt{S}\gamma$$

con tale associazione dà luogo alla premessa

$$\texttt{abaSbb} = \delta\texttt{S}\gamma$$

e ciò suggerisce di estendere la sostituzione mediante le associazioni $\texttt{aba}/\delta$ e $\texttt{bb}/\gamma$. A questo punto non abbiamo altra scelta che completare la sostituzione con l'associazione $\texttt{abaabbb}/\beta$, se vogliamo garantire che tutte le premesse siano soddisfatte. Riassumendo, abbiamo così determinato la sostituzione

$$\vartheta = \{\texttt{abaSbb}/\alpha, \texttt{abaabbb}/\beta, \texttt{aba}/\delta, \texttt{bb}/\gamma\}$$

[3] $\gamma \not\rightarrow \gamma'$ indica il fatto che $(\gamma, \gamma') \notin \rightarrow$.

in modo che tutte le premesse della regola $(r2)\vartheta$ siano soddisfatte. Con tale sostituzione, secondo la Definizione 2.9, il sistema contiene la transizione $\texttt{abaSbb} \rightarrow \texttt{abaabaa}$.

Ragionando in modo del tutto analogo possiamo costruire la sostituzione

$$\vartheta' = \{\texttt{abaSbb}/\alpha, \texttt{abaaSbbb}/\beta, \texttt{aba}/\delta, \texttt{bb}/\gamma\}$$

per concludere, grazie all'istanza $(r2)\vartheta'$, l'esistenza della transizione $\texttt{abaSbb} \rightarrow \texttt{abaaSbbb}$.

2.2 Esempi di sistemi di transizioni

2.2.1 Grammatiche come sistemi di transizioni

Nel paragrafo precedente, data una grammatica libera G abbiamo definito un sistema di transizioni S_G la cui relazione di transizione corrisponde esattamente alla nozione di *derivazione* nella grammatica, nel senso che $\alpha \rightarrow_G \beta$ se e solo se la stringa β è ottenibile dalla stringa α riscrivendo un simbolo non terminale di α con la parte destra di una produzione per quel simbolo. Nell'esempio che segue, vediamo come definire un nuovo sistema di transizioni a partire da una grammatica, in cui però la relazione di transizione corrisponde a riscrivere sempre il simbolo non terminale più a sinistra nella stringa.

Esempio 2.10 Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera da contesto e sia S'_G il sistema di transizioni così definito.

- le configurazioni di S'_G sono stringhe in $(\Lambda \cup V)^*$;
- le configurazioni terminali di S'_G sono le stringhe in Λ^*;
- la relazione di transizione $\rightsquigarrow_G$ è definita dalle seguenti regole condizionali, una per ogni produzione $(A ::= \eta) \in P$:

$$\frac{\delta \in \Lambda^* \qquad \gamma \in (\Lambda \cup V)^*}{\delta A \gamma \rightsquigarrow_G \delta \eta \gamma}$$

Anche in questo caso è facile convincersi che $\alpha \in \mathcal{L}(G)$ se e solo se $S \overset{*}{\rightsquigarrow}_G \alpha$. ■

2.2.2 Un sistema di transizioni per la somma di numeri naturali

Vediamo un esempio di sistema di transizioni per il calcolo della somma tra numeri naturali. Questi ultimi sono rappresentati dalla seguente grammatica:

```
Num ::= Cif | Num Cif
Cif ::= 0 | 1 | 2 | ...  | 9
```

Nella definizione del sistema di transizioni utilizziamo un approccio di tipo gerarchico: definiamo cioè, dapprima, le transizioni di un *sottosistema* che definisce la somma tra cifre decimali con riporto e successivamente quelle del sistema per la somma di due numeri arbitrari. Quest'ultimo farà uso del sottosistema già definito laddove ciò risulti utile. Definiamo quindi due sistemi di transizioni:

- $\texttt{S}_{cr}$, che definisce la somma tra cifre decimali con riporto; la relazione di transizione di questo sottosistema è rappresentata dal simbolo $\rightarrow_{cr}$;
- $\texttt{S}_{n+}$, che definisce la somma tra numeri decimali; la relazione di transizione di questo sottosistema è rappresentata dal simbolo $\rightarrow_{n+}$.

Nel seguito useremo le seguenti convenzioni:

- $\texttt{c}$, $\texttt{c}'$, $\texttt{c}''$, ... stanno per generiche cifre decimali, ovvero elementi della categoria sintattica `Cif`;
- $\texttt{r}$, $\texttt{r}'$, $\texttt{r}''$, ... stanno per elementi dell'insieme $\{\texttt{0,1}\}$, i riporti derivanti dalla somma tra due cifre;
- $\texttt{n}$, $\texttt{n}'$, $\texttt{m}$, ... stanno per generici numeri decimali, ovvero elementi della categoria sintattica `Num`.

Per rendere più snella la trattazione, indicheremo con $\texttt{x} \in \texttt{T}$ il fatto che la stringa $\texttt{x}$ appartiene al linguaggio associato alla categoria sintattica $\texttt{T}$: scriveremo, ad esempio, $\texttt{0} \in \texttt{Cif}$, $\texttt{n} \in \texttt{Num}$ e così via.

$\mathtt{S}_{cr}$: somma tra cifre decimali con riporto

Le configurazioni del sistema di transizioni $\mathtt{S}_{cr}$ per la somma di due cifre sono di due tipi:

- $\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle$, che rappresentano lo stato in cui il sistema si appresta a sommare le due cifre $\mathtt{c}$ e $\mathtt{c}'$ e il riporto $\mathtt{r}$;
- $\langle \mathtt{c}, \mathtt{r} \rangle$, che rappresentano la cifra ed il riporto risultante dalla somma di due cifre. Sono queste, dunque, tutte e sole le configurazioni terminali.

Più formalmente:

$$\begin{aligned} \Gamma_{cr} &= \{\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle \mid \mathtt{r} \in \{0,1\} \text{ e } \mathtt{c}, \mathtt{c}' \in \mathtt{Cif}\} \cup \\ &\quad \{\langle \mathtt{c}, \mathtt{r} \rangle \mid \mathtt{r} \in \{0,1\} \text{ e } \mathtt{c} \in \mathtt{Cif}\} \\ \mathtt{T}_{cr} &= \{\langle \mathtt{c}, \mathtt{r} \rangle \mid \mathtt{r} \in \{0,1\} \text{ e } \mathtt{c} \in \mathtt{Cif}\}. \end{aligned}$$

La relazione di transizione è data dalle seguenti regole:

$\langle 0,0,0 \rangle \to_{cr} \langle 0,0 \rangle$	$\langle 0,0,1 \rangle \to_{cr} \langle 1,0 \rangle$
$\langle 0,1,0 \rangle \to_{cr} \langle 1,0 \rangle$	$\langle 0,1,1 \rangle \to_{cr} \langle 2,0 \rangle$
$\langle 0,2,0 \rangle \to_{cr} \langle 2,0 \rangle$	$\langle 0,2,1 \rangle \to_{cr} \langle 3,0 \rangle$
$\ldots \quad \ldots$	$\ldots \quad \ldots$
$\langle 9,8,0 \rangle \to_{cr} \langle 7,1 \rangle$	$\langle 9,8,1 \rangle \to_{cr} \langle 8,1 \rangle$
$\langle 9,9,0 \rangle \to_{cr} \langle 8,1 \rangle$	$\langle 9,9,1 \rangle \to_{cr} \langle 9,1 \rangle$

Il "calcolo", mediante il sistema appena definito, della somma di due cifre $\mathtt{c}$ e $\mathtt{c}'$ e di un riporto $\mathtt{r}$ corrisponde ad individuare una derivazione che porta la configurazione *iniziale* $\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle$ in una configurazione terminale $\langle \mathtt{c}'', \mathtt{r}' \rangle$. Si noti che, per come è definita $\to_{cr}$, tali derivazioni sono di lunghezza unitaria.

$\mathtt{S}_{n+}$: somma tra numeri decimali

Le configurazioni del sistema di transizioni $\mathtt{S}_{n+}$ per la somma di due numeri decimali sono di tre tipi:

- $\langle \mathtt{n}, \mathtt{n}' \rangle$, che rappresentano lo stato in cui il sistema si appresta a calcolare la somma tra $\mathtt{n}$ e $\mathtt{n}'$ e che chiameremo configurazioni *iniziali*;
- $\langle \mathtt{n}, \mathtt{n}', \mathtt{m}, \mathtt{r} \rangle$ che rappresentano uno stadio intermedio del calcolo in cui il sistema si appresta a sommare i numeri $\mathtt{n}$ e $\mathtt{n}'$ ed il riporto $\mathtt{r}$, avendo calcolato il risultato parziale $\mathtt{m}$;
- $\mathtt{n}$, che rappresentano il risultato finale del calcolo; queste sono dunque tutte e sole le configurazioni terminali.

Più formalmente:

$$\begin{aligned} \Gamma_{n+} &= \{\langle \mathtt{n}, \mathtt{n}' \rangle \mid \mathtt{n}, \mathtt{n}' \in \mathtt{Num}\} \cup \\ &\quad \{\langle \mathtt{n}, \mathtt{n}', \mathtt{m}, \mathtt{r} \rangle \mid \mathtt{r} \in \{0,1\} \text{ e } \mathtt{n}, \mathtt{n}', \mathtt{m} \in \mathtt{Num}\} \cup \\ &\quad \{\mathtt{n} \mid \mathtt{n} \in \mathtt{Num}\} \\ \mathtt{T}_{n+} &= \{\mathtt{n} \mid \mathtt{n} \in \mathtt{Num}\} \end{aligned}$$

Il sistema di transizioni $\mathtt{S}_{n+}$ utilizza $\mathtt{S}_{cr}$ come sottosistema, e ciò è reso esplicito dal fatto che alcune transizioni in $\to_{n+}$ hanno come prerequisito una derivazione in $\to_{cr}$. In effetti, le transizioni di $\to_{n+}$ corrispondono al calcolo della somma tra numeri che abbiamo imparato fin dalle scuole elementari, che prevede la somma di due cifre e di un riporto come operazione "elementare". Consideriamo come esempio il procedimento che un alunno delle scuole elementari segue per effettuare la somma di 54 e 38:

- stadio iniziale del calcolo

```
 54
 38
```

A questa situazione corrisponde la configurazione iniziale $\langle \texttt{54}, \texttt{38} \rangle$.

- stadio intermedio

```
 1
 54
 38
----
 2
```

poiché `4+8=2` con riporto `1`.
In questa situazione intermedia il calcolo ha prodotto un risultato parziale, `2`, ed un riporto, `1`, sommando le prime due cifre dei numeri di partenza; queste ultime, di qui in poi, possono essere semplicemente ignorate. A questa situazione corrisponde la transizione $\langle \texttt{54}, \texttt{38} \rangle \rightarrow_{n+} \langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle$, dove `5` e `3` rappresentano i numeri ancora da sommare, `2` il risultato intermedio e `1` il riporto.

- stadio intermedio

```
0
 54
 38
----
 92
```

poiché `5+3+1=9` con riporto `0`.
A questa situazione corrisponde la transizione $\langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle \rightarrow_{n+} \langle \texttt{0}, \texttt{0}, \texttt{92}, \texttt{0} \rangle$.

- risultato della somma: `92`.
Non essendoci più cifre da sommare, il risultato è `92`. A questa situazione corrisponde la transizione $\langle \texttt{0}, \texttt{0}, \texttt{92}, \texttt{0} \rangle \rightarrow_{n+} \texttt{92}$.

Si noti come, nei due stadi intermedi del calcolo, abbiamo assunto che la somma di due cifre e di un riporto sia nota all'alunno. Nel caso del sistema di transizioni, ciò corrisponde ad assumere che derivazioni in $\rightarrow_{cr}$ possano essere prerequisiti per le transizioni di $\rightarrow_{n+}$. Formalmente ciò può essere espresso mediante regole condizionali. Ad esempio, la transizione corrispondente al primo dei due stadi intermedi, si ottiene grazie alla regola condizionale:

$$\frac{\langle \texttt{4}, \texttt{8}, \texttt{0} \rangle \rightarrow_{cr} \langle \texttt{2}, \texttt{1} \rangle}{\langle \texttt{54}, \texttt{38} \rangle \rightarrow_{n+} \langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle}$$

Si noti come la premessa sia definita in termini del (sotto-)sistema di transizioni $\mathtt{S}_{cr}$.

Le transizioni di $\rightarrow_{n+}$ sono date qui di seguito:

$$\frac{\langle \mathtt{c}, \mathtt{c}', \texttt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{c}, \mathtt{c}' \rangle \rightarrow_{n+} \langle \texttt{0}, \texttt{0}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(i)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \texttt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{nc}, \mathtt{c}' \rangle \rightarrow_{n+} \langle \mathtt{n}, \texttt{0}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(ii)}$$

$$\frac{\langle \mathtt{c}', \mathtt{c}, \texttt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{c}', \mathtt{nc} \rangle \rightarrow_{n+} \langle \texttt{0}, \mathtt{n}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(iii)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', 0\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}'\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}'', \mathtt{r}\rangle} \quad \text{(iv)}$$

$$\langle 0, 0, \mathtt{m}, 0\rangle \rightarrow_{n+} \mathtt{m} \quad \text{(v)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{c}, \mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle 0, 0, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{se } \langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \neq \langle 0, 0, 0\rangle \quad \text{(vi)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{nc}, \mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{n}, 0, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{(vii)}$$

$$\frac{\langle \mathtt{c}', \mathtt{c}, \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{c}', \mathtt{nc}, \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle 0, \mathtt{n}, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{(viii)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{(ix)}$$

Osserviamo come le precedenti regole sono in realtà *schemi di regole*: ogni schema rappresenta una collezione di regole (che chiameremo *istanze* della regola), ciascuna ottenuta rimpiazzando i simboli dello schema con particolari valori. Ad esempio, (ii) rappresenta, tra le altre, le regole:

$$\frac{\langle 9, 5, 0\rangle \rightarrow_{cr} \langle 4, 1\rangle}{\langle 379, 5\rangle \rightarrow_{n+} \langle 37, 0, 4, 1\rangle} \qquad \frac{\langle 2, 4, 0\rangle \rightarrow_{cr} \langle 6, 0\rangle}{\langle 82, 4\rangle \rightarrow_{n+} \langle 8, 0, 6, 0\rangle}$$

Le transizioni (i) ÷ (iv) corrispondono ai passi iniziali del calcolo in cui i numeri da sommare sono costituiti: entrambi da una sola cifra, il primo da più cifre e il secondo da una sola cifra, il primo da una sola cifra ed il secondo da più cifre ed entrambi da più cifre, rispettivamente. La transizione (v) corrisponde invece al passo terminale del calcolo: la configurazione $\langle 0, 0, \mathtt{m}, 0\rangle$ rappresenta la situazione in cui non ci sono più cifre da sommare, non c'è riporto ed il risultato intermedio è $\mathtt{m}$. Le regole (vi) ÷ (ix) corrispondono ai passi intermedi del calcolo in cui il risultato intermedio è $\mathtt{m}$ ed i numeri ancora da sommare sono costituiti: entrambi da una sola cifra, il primo da più cifre e il secondo da una sola cifra, il primo da una sola cifra ed il secondo da più cifre ed entrambi da più cifre, rispettivamente. La condizione a margine della regola (vi) inibisce l'uso della transizione stessa a partire dalla configurazione $\langle 0, 0, \mathtt{m}, 0\rangle$, per la quale è applicabile la regola (v).

Esempio 2.11 Vediamo un esempio di derivazione nel sistema che abbiamo appena definito, che corrisponde al "calcolo" della somma tra 927 e 346. Ad ogni passo della derivazione associamo (tra parentesi graffe) la giustificazione del passo stesso, costituita dall'etichetta che identifica la regola utilizzata e dall'istanza della sua eventuale premessa:

$$\begin{aligned} & \langle 927, 346\rangle \\ \rightarrow_{n+} \quad & \{\text{(iv)}, \langle 7, 6, 0\rangle \rightarrow_{cr} \langle 3, 1\rangle\ \} \\ & \langle 92, 34, 3, 1\rangle \end{aligned}$$

$\rightarrow_{n+}$ $\{(\text{ix}),\ \langle 2,4,1\rangle \rightarrow_{cr} \langle 7,0\rangle\ \}$

$\langle 9,3,73,0\rangle$

$\rightarrow_{n+}$ $\{(\text{vi}),\ \langle 9,3,0\rangle \rightarrow_{cr} \langle 2,1\rangle\ \}$

$\langle 0,0,273,1\rangle$

$\rightarrow_{n+}$ $\{(\text{vi}),\ \langle 0,0,1\rangle \rightarrow_{cr} \langle 1,0\rangle\ \}$

$\langle 0,0,1273,0\rangle$

$\rightarrow_{n+}$ $\{(\text{v})\}$

1273

■

2.2.3 Rappresentazione e semantica dei numeri

Nel paragrafo precedente abbiamo usato la comune rappresentazione dei numeri naturali mediante sequenze di cifre decimali. Per capire in che senso il sistema $\mathtt{S}_{n+}$ *definisce* la somma tra numeri naturali, dobbiamo introdurre la distinzione tra un numero naturale e la sua rappresentazione. Sia allora $\mathbb{N} = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \ldots\}$ l'insieme dei numeri naturali, sul quale sono definite le usuali operazioni di somma ($+$), prodotto ($\times$), quoziente e resto della divisione intera (*div* e *mod*): assumeremo anche di avere le operazioni di "uguaglianza" ($=$), "disuguaglianza" ($\neq$), "minore"($\prec$), "maggiore" ($\succ$), "maggiore o uguale" ($\succeq$), "minore o uguale" ($\preceq$) su coppie di valori in $\mathbb{N}$[4]. Ad ogni elemento `n` di `Num` corrisponde il *valore* in $\mathbb{N}$ di cui `n` è la *rappresentazione*. Tale associazione è data dalla seguente *funzione di valutazione* $\eta : \mathtt{Num} \rightarrow \mathbb{N}$, definita per casi nel modo seguente:

$$\begin{array}{rcl} \eta(\mathtt{0}) & = & \underline{0} \\ \eta(\mathtt{1}) & = & \underline{1} \\ & \ldots & \\ \eta(\mathtt{9}) & = & \underline{9} \\ \eta(\mathtt{nc}) & = & (\eta(\mathtt{n}) \ \times\ \underline{10}) \ +\ \eta(\mathtt{c}) \end{array}$$

Si noti come i casi nella definizione di η corrispondano proprio ai casi della definizione sintattica di `Num`. Analogamente, è possibile definire la *funzione di rappresentazione* $\nu : \mathbb{N} \rightarrow \mathtt{Num}$, definita come segue:

$$\begin{array}{rcll} \nu(\underline{0}) & = & \mathtt{0} & \\ \nu(\underline{1}) & = & \mathtt{1} & \\ & \ldots & & \\ \nu(\underline{9}) & = & \mathtt{9} & \\ \nu(\underline{\mathtt{n}}) & = & \nu(\underline{\mathtt{n}}\ div\ \underline{10})\nu(\underline{\mathtt{n}}\ mod\ \underline{10}) & \text{se } \underline{\mathtt{n}} \succ \underline{9} \end{array}$$

La scelta di denotare i *valori* in $\mathbb{N}$ con la notazione $\underline{\mathtt{n}}$ non deve indurre a pensare che la funzione di valutazione (risp. rappresentazione) possa essere definita semplicemente come $\eta(\mathtt{n}) = \underline{\mathtt{n}}$ (risp. $\nu(\underline{\mathtt{n}}) = \mathtt{n}$). La notazione $\underline{\mathtt{n}}$ è anch'essa una *rappresentazione*, diversa da quella sintattica, da noi scelta per denotare un *valore* in $\mathbb{N}$. La funzione di valutazione definita per casi fornisce un modo per calcolare in modo *costruttivo* il valore corrispondente ad una stringa della particolare sintassi (linguaggio di rappresentazione) utilizzata (nel nostro caso, di una stringa di `Num`).

Per convincerci ulteriormente della necessità di distinguere tra *valori* e *rappresentazioni*, utilizziamo una sintassi diversa per la descrizione di numeri naturali, che corrisponde alla rappresentazione *binaria* dei

[4]Si noti la necessità di distinguere tra i simboli utilizzati per denotare le operazioni su $\mathbb{N}$ ed i simboli utilizzati per rappresentare sintatticamente gli stessi. Così, ad esempio, `+` è il simbolo *sintattico* utilizzato per rappresentare l'operazione di somma $+$ tra numeri naturali, ovvero tra elementi di $\mathbb{N}$. Analogamente, $\times, -, div, mod, =, \neq, \prec, \succ, \succeq, \preceq$ sono operazioni su coppie di naturali e non vanno confuse con i corrispondenti simboli sintattici `*`, `-`, `/`, `%`, `==`, `!=`, `<`, `>`, `>=` e `<=`.

numeri. Utilizziamo volutamente i simboli `z` e `u`, anziché gli usuali simboli `0` e `1`, per le cifre binarie (*bit*) proprio per evidenziarne il carattere puramente *simbolico*.

```
Binary ::= Bit | Binary Bit
Bit ::= z | u
```

Qual è il numero naturale rappresentato dalla stringa `zuzz`? Definiamo per casi una nuova funzione di valutazione $\eta' : \texttt{Binary} \to I\!N$ per il nuovo linguaggio. Nella definizione di η' utilizziamo la convenzione di denotare con x una generica stringa in `Binary` e con b un generico bit (ovvero b sta per `z` oppure `u`).

$$\begin{aligned} \eta'(\texttt{z}) &= \underline{0} \\ \eta'(\texttt{u}) &= \underline{1} \\ \eta'(xb) &= (\eta'(x) \times \underline{2}) + \eta'(b) \end{aligned}$$

Applicando la funzione η' alla stringa `zuzz` otteniamo:

$$\begin{aligned} \eta'(\texttt{zuzz}) &= (\eta'(\texttt{zuz}) \times \underline{2}) + \eta'(\texttt{z}) \\ &= (\eta'(\texttt{zuz}) \times \underline{2}) + \underline{0} \\ &= (((\eta'(\texttt{zu}) \times \underline{2}) + \eta'(\texttt{z})) \times \underline{2}) + \underline{0} \\ &= (((\eta'(\texttt{zu}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\ &= (((((\eta'(\texttt{z}) \times \underline{2}) + \eta'(\texttt{u})) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\ &= (((((\underline{0} \times \underline{2}) + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\ &= ((((\underline{0} + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\ &= (((\underline{1} \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\ &= ((\underline{2} + \underline{0}) \times \underline{2}) + \underline{0} \\ &= (\underline{2} \times \underline{2}) + \underline{0} \\ &= \underline{4} + \underline{0} \\ &= \underline{4} \end{aligned}$$

Il sistema di transizioni $\texttt{S}_{n+}$ definisce la somma tra numeri naturali nel senso che, data la rappresentazione di due valori naturali, fornisce la rappresentazione della loro somma. Ciò può essere espresso formalmente dal seguente asserto:

$$\textbf{se } \langle \texttt{n}, \texttt{m} \rangle \to^*_{n+} \texttt{k} \qquad \textbf{allora} \qquad \eta(\texttt{n}) + \eta(\texttt{m}) = \eta(\texttt{k})$$

È facile convincersi che vale anche il viceversa, ovvero:

$$\textbf{se } \underline{\texttt{n}} + \underline{\texttt{m}} = \underline{\texttt{k}} \qquad \textbf{allora} \qquad \langle \nu(\underline{\texttt{n}}), \nu(\underline{\texttt{m}}) \rangle \to^*_{n+} \nu(\underline{\texttt{k}})$$

Sebbene la distinzione tra valori/operazioni e loro rappresentazione sintattica sia formalmente necessaria, nel seguito la ometteremo quando ciò non crei confusione o ambiguità. Utilizzeremo ad esempio `n` sia per indicare il valore $\underline{\texttt{n}}$ sia per indicare la rappresentazione di $\underline{\texttt{n}}$, così come scriveremo $\underline{\texttt{n}} > \underline{\texttt{m}}$ o `n`>`m` anziché $\underline{\texttt{n}} \succ \underline{\texttt{m}}$.

2.3 Altri esempi

In questo paragrafo consideriamo un semplice linguaggio e definiamo varie semantiche per le stringhe del linguaggio stesso attraverso la definizione di diversi sistemi di transizioni. Il linguaggio, che indicheremo con $\mathcal{L}$ è quello delle stringhe non vuote formate dai simboli `a` e `b`, ovvero $\mathcal{L} = (\{\texttt{a}, \texttt{b}\})^+$, che può essere definito dalla seguente grammatica

```
A ::= aA | bA | a | b
```

Esempio 2.12 La prima semantica che vogliamo definire è la seguente: il significato di una stringa $\alpha \in \mathcal{L}$ è il numero naturale dato dal numero di occorrenze del simbolo `a` in α. Quindi, ad esempio, le stringhe `aabab` e `aaa` hanno come semantica il numero naturale 3, così come ogni stringa formata da sole `b` rappresenta il numero naturale 0, e così via. Per esprimere tale semantica mediante un sistema di transizioni $S_1 = \langle \Gamma_1, T_1, \rightarrow_1 \rangle$ possiamo procedere come segue:

- Γ_1 dovrà contenere, tra le altre, configurazioni che corrispondono a stringhe del linguaggio, ovvero l'insieme $\{\alpha | \alpha \in \mathcal{L}\}$
- T_1 dovrà essere costituito dall'insieme dei numeri naturali $\mathbb{N}$
- la relazione di transizione dovrà consentire derivazioni del tipo $\alpha \rightarrow_1^* \underline{\mathtt{n}}$ se e soltanto se il simbolo `a` occorre esattamente $\underline{\mathtt{n}}$ volte in α.

Un primo approccio consiste nel definire un sistema la cui relazione di transizione descrive i singoli passi di calcolo che, a partire da una stringa iniziale α, porta alla determinazione dell'opportuna configurazione finale. Gli stadi intermedi di tale calcolo vengono descritti tramite un ulteriore tipo di configurazioni, definite dal seguente insieme:

$$\{\langle \beta, \underline{\mathtt{m}} \rangle \mid \beta \in \mathcal{L}, \underline{\mathtt{m}} \in \mathbb{N}\}$$

Il significato di una configurazione $\langle \beta, \underline{\mathtt{m}} \rangle$ è il seguente:

- β rappresenta la porzione della stringa iniziale che deve essere ancora analizzata;
- $\underline{\mathtt{m}}$ rappresenta il numero di occorrenze di `a` nella porzione della stringa iniziale già analizzata.

Più precisamente, la relazione di transizione dovrà consentire derivazioni del tipo:

$$\alpha \rightarrow_1^* \langle \beta, \underline{\mathtt{k}} \rangle$$

se e soltanto se $\alpha = \beta'\beta$ e il numero di occorrenze di `a` in β' è $\underline{\mathtt{k}}$. Per definire la relazione di transizione, utilizziamo, quando necessario, un approccio *guidato dalla sintassi*, in cui le stringhe vengono analizzate per casi a seconda della loro struttura sintattica. Tale definizione deve garantire che le proprietà sopra enunciate di $\rightarrow_1$ siano soddisfatte. Nella scrittura delle regole condizionali che definiscono $\rightarrow_1$ utilizziamo per brevità la convenzione di denotare con $\alpha, \beta, \ldots$ stringhe del linguaggio.

$$\frac{\alpha = \mathtt{a}\beta \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \langle \beta, \underline{\mathtt{m}} \rangle} \qquad \text{(r1)}$$

$$\frac{\alpha = \mathtt{b}\beta}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \langle \beta, \underline{\mathtt{k}} \rangle} \qquad \text{(r2)}$$

$$\frac{\alpha = \mathtt{a} \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \underline{\mathtt{m}}} \qquad \text{(r3)}$$

$$\frac{\alpha = \mathtt{b}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \underline{\mathtt{k}}} \qquad \text{(r4)}$$

$$\frac{}{\alpha \rightarrow_1 \langle \alpha, \underline{\mathtt{0}} \rangle} \qquad \text{(r5)}$$

È facile convincersi che queste regole definiscono correttamente la relazione di transizione $\rightarrow_1$. Un approccio alternativo consiste nel definire $\rightarrow_1$ utilizzando un approccio *induttivo* che astrae dai singoli passi di calcolo e definisce direttamente la relazione tra le configurazioni iniziali e le configurazioni finali, ovvero tra una stringa e la sua semantica. La nuova definizione si basa sulle seguenti considerazioni informali:

- il numero di occorrenze di `a` nella stringa `a` (risp. `b`) è $\underline{1}$ (risp. $\underline{0}$).
- il numero di occorrenze di `a` nella stringa `a`α (risp. `b`α) è $\underline{n}+\underline{1}$ (risp. $\underline{n}$) se il numero di occorrenze di `a` in α è $\underline{n}$.

La traduzione di queste regole informali produce il seguente insieme di regole condizionali:

$$\frac{\alpha \rightarrow_1 \underline{k} \qquad \underline{n} = \underline{k}+\underline{1}}{\texttt{a}\alpha \rightarrow_1 \underline{n}} \qquad \text{(r1')}$$

$$\frac{\alpha \rightarrow_1 \underline{k}}{\texttt{b}\alpha \rightarrow_1 \underline{k}} \qquad \text{(r2')}$$

$$\frac{}{\texttt{a} \rightarrow_1 \underline{1}} \qquad \text{(r3')}$$

$$\frac{}{\texttt{b} \rightarrow_1 \underline{0}} \qquad \text{(r4')}$$

Si noti che, in questo caso, le regole condizionali contengono premesse che fanno riferimento alla relazione di transizione stessa. Come già accennato, il sistema che si ottiene in questo modo è più *astratto* del precedente: non si considerano, in questo approccio, configurazioni intermedie che descrivono stadi intermedi del calcolo, ma si definisce direttamente la relazione tra le configurazioni di partenza e le configurazioni terminali. Invece, nel caso delle regole (r1) ÷ (r5) date in precedenza, le transizioni definiscono i singoli passi del calcolo. Si rimanda alla prossima Sezione per un breve confronto tra i due approcci, detti, rispettivamente, semantica di valutazione e semantica naturale.

Esempio 2.13 Definiamo ora una semantica diversa per il linguaggio $\mathcal{L}$: il significato di una stringa α è il numero naturale $\underline{n}$ se e soltanto se $\alpha = \texttt{a}^{\underline{n}}\texttt{b}^{\underline{k}}$, con $\underline{n}, \underline{k} \geq \underline{0}$. Si noti che in questa semantica non tutte le stringhe di $\mathcal{L}$ hanno un significato: ad esempio, mentre il significato di `aaabb` è $\underline{3}$, la stringa `ababa` non ha alcun significato. Anche nel sistema di transizioni $S_2 = \langle \Gamma_2, T_2, \rightarrow_2 \rangle$ che ci accingiamo a definire avremo che Γ_2 contiene configurazioni *iniziali* del tipo α con $\alpha \in \mathcal{L}$ e che T_2 è l'insieme dei numeri naturali. Questa volta la relazione di transizione dovrà consentire sia il calcolo del numero di occorrenze di `a` sia la verifica che la stringa data ha una struttura opportuna. A questo scopo introduciamo configurazioni "intermedie" del tipo $\langle \beta, \underline{m}, x \rangle$ in cui le componenti β e $\underline{m}$ rappresentano rispettivamente (come nell'esempio precedente) la porzione di stringa che rimane da analizzare ed il numero di occorrenze di `a` presenti nella porzione di stringa già analizzata, mentre x segnala l'ultimo simbolo analizzato. In questo modo possiamo fare in modo che configurazioni del tipo $\langle \texttt{a}\alpha, \underline{n}, \texttt{b} \rangle$ siano *bloccate* in quanto corrispondono a stringhe che non hanno semantica.

Le regole condizionali che definiscono $\rightarrow_2$ sono le seguenti:

$$\frac{\underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \mathtt{a}\beta, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{m}}, \mathtt{a}\rangle} \qquad \text{(r1)}$$

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle} \qquad \text{(r2)}$$

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, \mathtt{b}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle} \qquad \text{(r3)}$$

$$\frac{\underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \mathtt{a}, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r4)}$$

$$\frac{}{\langle \mathtt{b}, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r5)}$$

Per quanto concerne le transizioni relative alle configurazioni iniziali del tipo α, possiamo introdurre una sola regola condizionale

$$\frac{}{\alpha \rightarrow_2 \langle \alpha, \underline{\mathtt{0}}, \mathtt{a}\rangle} \qquad \text{(r6)}$$

Si noti che in questa regola la terza componente della configurazione di arrivo non corrisponde al suo significato informale descritto in precedenza, ovvero non è l'ultimo simbolo analizzato. Tuttavia, questa formulazione della regola ci evita di prevedere quattro regole distinte corrispondenti ai quattro casi sintattici per α. Osserviamo infine che le regole (r2) e (r3) possono essere inglobate in una sola regola

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, x\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle}$$

In questo esempio, l'approccio astratto non è immediato come nel caso dell'esempio precedente. Ne diamo qui di seguito una possibile versione.

$$\frac{\beta \rightarrow_2 \underline{\mathtt{k}} \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\mathtt{a}\beta \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r1')}$$

$$\frac{\beta \rightarrow_2 \underline{\mathtt{0}}}{\mathtt{b}\beta \rightarrow_2 \underline{\mathtt{0}}} \qquad \text{(r2')}$$

$$\frac{}{\mathtt{a} \rightarrow_2 \underline{\mathtt{1}}} \qquad \text{(r3')}$$

$$\frac{}{\mathtt{b} \rightarrow_2 \underline{\mathtt{0}}} \qquad \text{(r4')}$$

La premessa della regola (r2') assicura l'applicabilità della regola stessa solo nel caso in cui la stringa ancora da esaminare, β, non contiene il simbolo $\mathtt{a}$.

3 La definizione operazionale della semantica

In questo paragrafo vediamo come i sistemi di transizioni possano essere utilizzati per dare la semantica in stile operazionale di un semplice linguaggio: quello delle espressioni aritmetiche. L'approccio operazionale alla semantica dei linguaggi di programmazione prevede, come accennato in precedenza, la definizione del comportamento di un sistema (di transizioni) in corrispondenza dei costrutti del linguaggio. Nel caso del linguaggio `Exp` che stiamo considerando, le configurazioni rappresentano gli stati in cui il sistema si trova ad operare durante il calcolo di una espressione mentre le transizioni rappresentano il procedere del calcolo stesso. Le configurazioni terminali corrispondono poi agli stati finali, che rappresentano in qualche modo il *risultato* del calcolo. A seconda del livello di dettaglio che si sceglie per descrivere l'evolvere del calcolo si ottiene un sistema più o meno *astratto*.

Tradizionalmente, nella descrizione dei linguaggi di programmazione si hanno due tipi di sistema, che si differenziano proprio per il livello di dettaglio scelto nella descrizione dei singoli passi di calcolo. La semantica che si ottiene nei due casi viene detta, rispettivamente, *semantica di valutazione* (*small-step*) e *semantica naturale* (*big-step*). È importante notare che, al di là della loro definizione, le due semantiche sono equivalenti, nel senso che conducono a risultati equivalenti a meno di semplici trasformazioni.

Nella semantica naturale, l'idea è di descrivere direttamente in che modo si ottiene il risultato di un calcolo, astraendo quanto più possibile dai passi intermedi. Più precisamente, ciò che la relazione di transizione descrive è direttamente la corrispondenza tra le configurazioni iniziali e le configurazioni finali.

Nella semantica di valutazione lo scopo è di descrivere i singoli passi del calcolo. Ad esempio, nel sistema del paragrafo precedente, le configurazioni non terminali rappresentano gli stadi intermedi della valutazione di una addizione e quelle terminali il risultato. Il valore di una somma (o meglio la rappresentazione di tale valore) si ottiene attraverso una derivazione che, a partire da una configurazione iniziale corrispondente all'addizione stessa e passando attraverso una serie di stadi intermedi, determina la configurazione terminale che rappresenta il valore del risultato. In queste note ci concentreremo sulla semantica naturale dei linguaggi.

3.1 Espressioni semplificate

Consideriamo dapprima il linguaggio delle espressioni semplificate:

```
Exp ::= Num | Exp + Exp
```

Una descrizione informale, ma sicuramente vicina alla nostra intuizione, di come si calcola il valore di un'espressione in tale linguaggio, può essere data come segue:

(r1) il valore di un'espressione costituita da un numero `n` è il valore rappresentato da `n`, ovvero $\underline{n}$;

(r2) il valore di un'espressione del tipo `E+E'` si ottiene sommando i valori $\underline{n}$ e $\underline{m}$, dove $\underline{n}$ è il valore di `E` e $\underline{m}$ è il valore di `E'`.

Osserviamo innanzitutto che le due regole corrispondono alle due alternative nella definizione sintattica di `Exp`. La seconda regola suggerisce anche un modo per *calcolare* il valore di un'espressione complessa `E+E'`, che consiste nel calcolare il valore delle due sottoespressioni `E` ed `E'` per poi calcolarne la somma (assumendo, naturalmente, di saper calcolare la somma tra due numeri naturali).

Vediamo come queste due regole consentano di determinare, ad esempio, il valore dell'espressione `2+3`. Data la struttura sintattica di `2+3`, non possiamo che appellarci inizialmente alla regola (r2)

(a) il valore di `2+3` è $\underline{n} + \underline{m}$, dove $\underline{n}$ è il valore di `2` e $\underline{m}$ è il valore di `3`.

Applicando tale regola abbiamo dunque ridotto il problema alla determinazione dei due valori $\underline{n}$ e $\underline{m}$ e a questo scopo possiamo utilizzare la regola (r1), guidati ancora dalla struttura sintattica delle espressioni in gioco.

(b1) il valore di `2` è $\underline{2}$

(b2) il valore di 3 è $\underline{3}$.

Sappiamo inoltre che

(b3) la somma di $\underline{2}$ e $\underline{3}$ è $\underline{5}$

e mettendo insieme (a), (b1), (b2), (b3), possiamo concludere che il valore di 2+3 è $\underline{5}$.

Come possiamo riflettere l'intuizione espressa dalle regole precedenti nella descrizione di un sistema di transizioni? Immaginiamo, in prima approssimazione, di utilizzare un sistema le cui configurazioni sono semplicemente espressioni e di voler definire una relazione di transizione i cui elementi siano transizioni del tipo $\mathtt{E} \to \mathtt{n}$, dove E è un'espressione e n è la rappresentazione del suo valore. Una possibilità consiste nell'elencare un numero (infinito!) di transizioni del tipo

$$\mathtt{0+1} \to \mathtt{1} \qquad \mathtt{1+1} \to \mathtt{2} \qquad \mathtt{1+2+2} \to \mathtt{5} \qquad \ldots$$

Un sistema descritto in questo modo è troppo *astratto*, nel senso che non suggerisce alcun modo per "calcolare" un'espressione e dunque ottenerne il valore.

Riconsideriamo dapprima la regola (r2): nel sistema che stiamo cercando di definire, una transizione $\mathtt{E} \to \mathtt{n}$ esprime il fatto che il valore di E è il numero naturale rappresentato da n. Dunque, (r2) può essere vista come regola per determinare la transizione dalla configurazione del tipo $\mathtt{E+E'}$ alla configurazione che rappresenta il valore di $\mathtt{E+E'}$. Detto altrimenti, la regola (r2) può essere riformulata in termini del sistema di transizioni che stiamo definendo, nel seguente modo:

$$\textbf{se} \quad \mathtt{E} \to \mathtt{n} \quad \textbf{e} \quad \mathtt{E'} \to \mathtt{m} \qquad \textbf{allora} \qquad \mathtt{E} + \mathtt{E'} \to \mathtt{k}, \text{ dove } \mathtt{k=n+m} \qquad (\dagger)$$

Vediamo ora come si traduce in termini di transizioni la regola (r1). Il valore dell'espressione n è rappresentato da n stesso e dunque per le configurazioni del tipo n non abbiamo bisogno di introdurre alcuna transizione: esse sono proprio le configurazioni terminali. Questa scelta, apparentemente corretta, è in contrasto proprio con la regola (†) appena introdotta: come calcoliamo, ad esempio, il valore di 2+3? La regola (†) ci impone di determinare due transizioni del tipo $\mathtt{2} \to \mathtt{n}$ e $\mathtt{3} \to \mathtt{m}$, ma abbiamo appena stabilito che non vi sono transizioni la cui parte sinistra è la rappresentazione di un numero!

Un modo per ovviare a questo inconveniente ci è di nuovo suggerito dalla descrizione informale delle regole (r1) e (r2): in esse, infatti, la distinzione tra rappresentazioni e valori è esplicita e ciò consente di parlare del valore $\underline{\mathtt{n}}$ di un'espressione semplice n. Nel sistema di transizioni che stiamo definendo, ciò si riflette nel trattare esplicitamente i valori: a questo scopo, basta considerare un nuovo insieme di configurazioni che, oltre alle espressioni, contiene anche i valori, ovvero gli elementi di $\mathbb{N}$.

Riassumendo, abbiamo le seguenti configurazioni:

$$\begin{aligned} \Gamma &= \{\mathtt{E} \mid \mathtt{E} \in \mathtt{Exp}\} \cup \\ &\quad\ \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbb{N}\} \\ \mathtt{T} &= \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbb{N}\} \end{aligned}$$

Siamo in grado di descrivere le regole del sistema di transizioni per le espressioni semplificate. In esse, utilizziamo la notazione introdotta in precedenza per le regole condizionali, nonché la convenzione che n rappresenta un elemento della categoria sintattica Num, mentre E ed $\mathtt{E'}$ rappresentano elementi della categoria sintattica Exp.

$$\mathtt{n} \to \underline{\mathtt{n}} \qquad (r1)$$

$$\frac{\mathtt{E} \to \underline{\mathtt{n}} \qquad \mathtt{E'} \to \underline{\mathtt{m}} \qquad \underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}}{\mathtt{E+E'} \to \underline{\mathtt{k}}} \qquad (r2)$$

Si noti come la definizione delle regole sia *guidata* dalla sintassi del linguaggio: la prima regola definisce le transizioni per configurazioni costituite da un elemento della categoria sintattica `Num`, mentre la seconda definisce transizioni per espressioni in cui compare l'operatore `+`. La premessa $\underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}$ della seconda regola corrisponde all'ipotesi di saper calcolare la somma tra numeri naturali: questo calcolo può essere visto come una derivazione in un sottosistema simile a $\mathtt{S}_{n+}$ del Paragrafo 2.2.2. Le altre due premesse della regola (r2) sono infine transizioni nel sistema che stiamo definendo: esse corrispondono al calcolo delle sottoespressioni dell'espressione di partenza.

Esempio 3.1 Vediamo un esempio di derivazione in questo sistema che corrisponde a valutare l'espressione `2+3+8`, ovvero a determinare una transizione del tipo $\mathtt{2+3+8} \to \underline{\mathtt{m}}$ per qualche $\underline{\mathtt{m}}$, applicando opportunamente le regole (r1) e (r2).

`2+3+8`

$\to$ { (r2),
(r1): $\mathtt{2} \to \underline{\mathtt{2}}$
(d1): $\mathtt{3+8} \to \underline{\mathtt{11}}, \quad \underline{\mathtt{2}} + \underline{\mathtt{11}} = \underline{\mathtt{13}}$}

$\underline{\mathtt{13}}$

La derivazione corrisponde ad una applicazione della regola (r2). Le premesse della regola, opportunamente istanziate, sono riportate come parti della giustificazione: la prima, un'istanza della regola (r1), e la terza, il calcolo della somma di due numeri, sono sufficientemente semplici da non richiedere ulteriori giustificazioni. La seconda, invece, è la conclusione che si ottiene dalla sottoderivazione (d1) seguente:

(d1):

`3+8`

$\to$ { (r2),
(r1): $\mathtt{3} \to \underline{\mathtt{3}}$
(r1): $\mathtt{8} \to \underline{\mathtt{8}}, \quad \underline{\mathtt{3}} + \underline{\mathtt{8}} = \underline{\mathtt{11}}$}

$\underline{\mathtt{11}}$

■

In generale, una derivazione è dunque costituita da una derivazione *principale* e da tante *sottoderivazioni* quante sono le transizioni non banali del tipo $\gamma \to \gamma'$ che compaiono nelle sue giustificazioni. Anche le sottoderivazioni possono contenere transizioni di questo tipo nelle proprie giustificazioni, che andranno eventualmente giustificate mediante ulteriori sottoderivazioni.

Un modo compatto per rappresentare l'intera derivazione è di costruire un albero in cui un nodo e i suoi figli rappresentano, rispettivamente, la conclusione dell'applicazione di una regola e le istanze delle premesse di tale regola. La derivazione precedente, in questa rappresentazione grafica, risulta essere la seguente.

$$\dfrac{\dfrac{}{\mathtt{2} \to \underline{\mathtt{2}}}\,(r1) \quad \dfrac{\dfrac{}{\mathtt{3} \to \underline{\mathtt{3}}}\,(r1) \quad \dfrac{}{\mathtt{8} \to \underline{\mathtt{8}}}\,(r1) \quad \underline{\mathtt{11}} = \underline{\mathtt{3}} + \underline{\mathtt{8}}}{\mathtt{3+8} \to \underline{\mathtt{11}}}\,(r2) \quad \underline{\mathtt{13}} = \underline{\mathtt{2}} + \underline{\mathtt{11}}}{\mathtt{2+3+8} \to \underline{\mathtt{13}}}\,(r2)$$

L'albero è stato ottenuto partendo dalla espressione `2+3+8` da valutare ed applicando la regola (r2) ottenendo

$$\dfrac{\mathtt{2} \to \underline{\mathtt{n}} \quad \mathtt{3+8} \to \underline{\mathtt{m}} \quad \underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}}{\mathtt{2+3+8} \to \underline{\mathtt{k}}}\,(r2)$$

Le ulteriori applicazioni delle regole alle premesse hanno poi consentito di espandere l'albero verso l'alto e di determinare i valori $\underline{\texttt{n}}$, $\underline{\texttt{m}}$ e $\underline{\texttt{k}}$, fino ad ottenere l'albero finale rappresentato nella figura precedente. Un ulteriore passo consente ad esempio di determinare il valore $\underline{\texttt{n}}$ come segue:

$$\dfrac{\dfrac{}{\texttt{2} \rightarrow \underline{\texttt{2}}}\ (r1) \quad \texttt{3} + \texttt{8} \rightarrow \underline{\texttt{m}} \quad \underline{\texttt{k}} = \underline{\texttt{2}} + \underline{\texttt{m}}}{\texttt{2} + \texttt{3} + \texttt{8} \rightarrow \underline{\texttt{k}}}\ (r2)$$

Si noti che l'applicazione di regole senza premesse determina un nodo foglia (cioè senza figli) dell'albero.

In queste note, tuttavia, continueremo ad utilizzare prevalentemente la rappresentazione che rimanda il calcolo delle premesse a sottoderivazioni presentate successivamente.

Un'osservazione importante sull'esempio precedente riguarda la scelta delle sottoespressioni ridotte nelle giustificazioni. Per quale motivo abbiamo scelto di ridurre, nel primo passo della derivazione, la sottoespressione `3+8` e non, ad esempio, la sottoespressione `2+3`? Ricordiamo, a questo proposito, che la stringa `2+3+8` è per noi una rappresentazione lineare del suo albero di derivazione in una grammatica, non ambigua, che definisce lo stesso linguaggio delle espressioni associato alla grammatica, ambigua, data in precedenza. Tutte le scelte fatte nella derivazione precedente corrispondono all'assunzione che l'albero di derivazione della stringa `2+3+8` sia quello rappresentato (ancora in modo lineare, ma questa volta mettendo in evidenza la struttura dell'espressione) da `2+[3+8]`[5].

Esempio 3.2 Vediamo come esempio la derivazione che si ottiene assumendo che l'albero sintattico sia ora `[2+3]+8`.

`2+3+8`

→ { (r2),
(d1): $\texttt{2} + \texttt{3} \rightarrow \underline{\texttt{5}}$
(r1): $\texttt{8} \rightarrow \underline{\texttt{8}}$
$\underline{\texttt{5}} + \underline{\texttt{8}} = \underline{\texttt{13}}$ }

$\underline{\texttt{13}}$

con la sottoderivazione (d1) ottenuta come segue:

(d1):
`2+3`

→ { (r2),
(r1): $\texttt{2} \rightarrow \underline{\texttt{2}}$
(r1): $\texttt{3} \rightarrow \underline{\texttt{3}}$
$\underline{\texttt{2}} + \underline{\texttt{3}} = \underline{\texttt{5}}$ }

$\underline{\texttt{5}}$

Le proprietà dell'operatore di somma tra numeri naturali garantiscono che il risultato, nei due casi, è il medesimo, ma ciò chiaramente non vale per un linguaggio più espressivo come quello che stiamo per introdurre. ■

3.2 Le espressioni a valori naturali

La grammatica del linguaggio che definisce le espressioni a valori naturali è data dalle seguenti produzioni:

```
Exp ::= Num | (Exp) | Exp Op Exp.
Op  ::= + | - | * | / | %
Num ::= ...
```

[5] Le parentesi `[` e `]` non devono essere intese come simboli sintattici, ma solo come ausilio per rappresentare linearmente l'albero di derivazione.

Osserviamo che questa grammatica non solo non assegna nessuna precedenza agli operatori, ma è anche ambigua: ciò ha solamente lo scopo di semplificare la definizione semantica. Infatti, la semantica operazionale (o meglio le regole che definiscono la relazione di transizione) è definita per casi in accordo alla sintassi del linguaggio. Poiché il processo di disambiguazione e di assegnazione di precedenze tra gli operatori richiede l'introduzione di ulteriori simboli non terminali nella grammatica, questo significherebbe complicare anche la definizione delle regole a scapito della chiarezza. La possibilità di scrivere espressioni non ambigue è data dalla presenza della produzione `(Exp)`. È da notare inoltre che, concettualmente, la definizione semantica non risulterebbe diversa: per queste ragioni abbiamo scelto di mantenere la definizione più compatta possibile a scapito della completezza.

Denotiamo con $\rightarrow_{exp}$ la relazione di transizione del sistema $\mathtt{S}_{exp}$ che definisce la semantica del linguaggio per le espressioni a valori naturali, le cui configurazioni sono, come nel caso delle espressioni del paragrafo precedente:

$$\begin{array}{rcl} \Gamma_{exp} & = & \{\mathtt{E} \mid \mathtt{E} \in \mathtt{Exp}\} \cup \\ & & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in I\!N\} \\ \mathtt{T}_{exp} & = & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in I\!N\} \end{array}$$

Le regole che definiscono $\rightarrow_{exp}$ sono elencate qui di seguito. Alcune delle premesse utilizzate nelle regole corrispondono ad operazioni "elementari" tra numeri che si suppone di saper calcolare mediante un sottosistema opportuno (come nel caso della premessa $\underline{\mathtt{n}}+\underline{\mathtt{m}} = \underline{\mathtt{k}}$ della regola (r2) del paragrafo precedente). Per semplicità, tali premesse verranno rappresentate mediante uguaglianze del tipo $\underline{\mathtt{n}} + \underline{\mathtt{m}} = \underline{\mathtt{k}}$, $\underline{\mathtt{n}} \times \underline{\mathtt{m}} = \underline{\mathtt{k}}$, ecc.

$$\mathtt{n} \rightarrow_{exp} \underline{\mathtt{n}} \qquad (exp_n)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\mathtt{E}+\mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_+)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\mathtt{E}*\mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_*)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\mathtt{E}\text{-}\mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_-)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ \mathit{div}\ \underline{\mathtt{n}}'}{\mathtt{E}\ /\ \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{div})$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ \mathit{mod}\ \underline{\mathtt{n}}'}{\mathtt{E}\ \%\ \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{mod})$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}}}{(\mathtt{E}) \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{()})$$

Queste regole sono semplici estensioni delle regole date nel paragrafo precedente. Si osservi che le premesse delle regole (exp_{div}) e (exp_{mod}), che corrispondono al calcolo di quoziente e resto della divisione intera, contengono, tra le premesse, la condizione $\underline{\mathtt{n}}' \neq \underline{0}$, caso in cui l'operazione di divisione non è definita. Analogamente, la premessa $\underline{\mathtt{n}} \geq \underline{\mathtt{n}}'$ della regola (exp_-) è necessaria per garantire che il risultato della differenza sia ancora un valore naturale.

Un'ultima osservazione riguarda il significato della regola $(exp_{()})$: se letta distrattamente, essa può apparire scorretta, dal momento che la transizione consiste esclusivamente nella "semplificazione" di `(E)` in `E`. Tale semplificazione, se vista nel contesto di una espressione complessa, non sarebbe lecita: si pensi ad esempio alla semplificazione da `(2+3)*5` in `2+3*5` e alla usuale precedenza tra gli operatori `+` e `*` . Ricordiamo ancora che, in realtà, `E` sta ad indicare l'*albero sintattico* dell'espressione `E`: dunque la semplificazione precedente sta ad indicare che, nel contesto in cui si trova, l'albero sintattico di `(E)` viene rimpiazzato dall'albero sintattico di `E`, e dunque la regola è corretta. Utilizzando ancora le parentesi `[` e `]` per mettere in evidenza la struttura dell'albero di derivazione, la regola $(exp_{()})$ consente di ottenere la transizione $[(2+3)] * 5 \rightarrow_{exp} [2+3] * 5$.

In che senso questo sistema definisce la semantica del linguaggio `Exp`? L'idea è quella di vedere le configurazioni finali come i *valori* delle espressioni di partenza. Denotando con $\mathcal{E}[\![\mathtt{E}]\!]$ la semantica di un'espressione, abbiamo formalmente:

$$\boxed{\mathcal{E}[\![\mathtt{E}]\!] = \underline{\mathtt{k}} \quad \textbf{se e solo se} \quad \mathtt{E} \rightarrow_{exp} \underline{\mathtt{k}}.}$$

Si noti che nel sistema $\mathtt{S}_{exp}$ esistono configurazioni *bloccate*, quali ad esempio `2-15`, `5 / (3-3)`, e così via. Intuitivamente, queste configurazioni corrispondono ad espressioni che, seppure sintatticamente corrette, non hanno un valore naturale.

Esempio 3.3 Vediamo quale è la derivazione in $\mathtt{S}_{exp}$ che consente di stabilire che il valore dell'espressione `25 - 15 / 2` è $\underline{18}$.

`25 - 15 / 2`

$\rightarrow_{exp}$ { $(exp_{\_})$,
(exp_n): $\mathtt{25} \rightarrow_{exp} \underline{25}$,
(d1): `15 / 2` $\rightarrow_{exp} \underline{7}$,
$\underline{25} \geq \underline{7}, \quad \underline{25} - \underline{7} = \underline{18}$ }

$\underline{18}$

dove (d1) è data da:

(d1):

`15 / 2`

$\rightarrow_{exp}$ { (exp_{div}),
(exp_n): $\mathtt{15} \rightarrow_{exp} \underline{15}$,
(exp_n): $\mathtt{2} \rightarrow_{exp} \underline{2}$,
$\underline{2} \neq \underline{0}, \quad \underline{15}\ div\ \underline{2} = \underline{7}$ }

$\underline{7}$

Osserviamo ancora che le sottoderivazioni "semplici" (che corrispondono ad istanze della regola (exp_n)) sono state direttamente riportate nelle giustificazioni delle regole in cui vengono utilizzate. La sottoderivazione che corrisponde alla valutazione di `15 / 2` è stata invece indicata con (d1) e riportata esplicitamente. Osserviamo infine che anche in questo caso è stata fatta una precisa assunzione sull'albero sintattico dell'espressione in gioco (rappresentato da `25 - [15 /2]`). ■

Esercizi

1. Valutare le seguenti espressioni, indicando per ciascuna l'albero sintattico sottinteso nella derivazione:

   (a) `15+(20 / 6) - 15`

(b) `100 % 7 + 11`

(c) `35 - (10 % 4) + 7`

2. Valutare l'espressione `12 / 2 - 2` supponendo che il suo albero sintattico sia:

   (a) `[12 / 2] - 2`

   (b) `12 / [2 - 2]`

4 Lo stato

Immaginiamo di estendere il linguaggio delle espressioni del Paragrafo 3.2 aggiungendo la nuova produzione `Exp ::= Ide`, che consente di scrivere espressioni in cui compaiono dei nomi (identificatori)[6]. È evidente che il significato di un'espressione come `x+2` dipende dal valore associato al nome `x`: in generale, il valore di un'espressione in cui compaiono i nomi $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ dipende dal valore associato a tali nomi[7]. Per descrivere la semantica di espressioni siffatte, aggiungiamo un ulteriore ingrediente al nostro sistema di transizioni, il cosiddetto *stato*. In questa sezione introduciamo dapprima lo stato nella sua forma più semplice, ovvero come tabella (un insieme di associazioni tra nomi e valori), e successivamente in una forma più strutturata, ovvero come sequenza di tabelle, necessaria per trattare alcuni meccanismi di Java, quali l'annidamento di blocchi.

4.1 Il concetto di stato

Nella sua forma più semplice, uno stato non è altro che un insieme di associazioni del tipo $\langle nome, val \rangle$ dove *val* è un valore. Indichiamo genericamente con `Val` l'insieme dei valori associati ai nomi nello stato: dunque, per quanto visto sino ad ora `Val` è l'insieme $\mathbb{N}$. Uno stato φ può essere visto come una funzione

$$\varphi : \mathtt{Ide} \mapsto \mathtt{Val}$$

che, dato un nome `x` restituisce il valore associato ad `x` in φ. In generale, uno stato φ è una funzione *parziale* sul dominio `Ide`, ovvero una funzione definita solo su alcuni elementi di `Ide` e non su altri. Addirittura, ci troveremo sempre a trattare con stati che contengono associazioni per un numero molto limitato di identificatori (tipicamente quelli che compaiono in un programma) e dunque per un sottoinsieme molto ristretto dell'insieme (infinito!) `Ide`.

Nel seguito, utilizzeremo $\varphi, \varphi_0, \varphi', \varphi'', \ldots$ per denotare generici stati. Inoltre, adotteremo l'usuale notazione $\varphi(\mathtt{x})$ per denotare il valore *associato ad* `x` *nello stato* φ. Dato uno stato φ che non contiene alcuna associazione per l'identificatore `x`, diremo che $\varphi(\mathtt{x}) = \perp$, dove $\perp$ indica appunto che la funzione φ è indefinita sull'argomento `x`.

A volte avremo bisogno di indicare esplicitamente *tutti e soli* gli identificatori per i quali è prevista una associazione nello stato φ. Siano $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_k$ tali identificatori e siano $\underline{\mathtt{v}}_1, \underline{\mathtt{v}}_2, \ldots, \underline{\mathtt{v}}_k$ i valori ad essi associati nello stato φ; un modo per rappresentare tale stato consiste nel definire φ come la funzione:

$$\varphi(x) = \begin{cases} \underline{\mathtt{v}}_1 & \text{se } x = \mathtt{x}_1 \\ \underline{\mathtt{v}}_2 & \text{se } x = \mathtt{x}_2 \\ \ldots & \\ \underline{\mathtt{v}}_k & \text{se } x = \mathtt{x}_k \\ \perp & \text{altrimenti} \end{cases}$$

Un modo alternativo, ma più conciso, consiste nell'elencare le associazioni presenti in φ come segue:

$$\varphi = \{\mathtt{x}_1 \mapsto \underline{\mathtt{v}}_1, \mathtt{x}_2 \mapsto \underline{\mathtt{v}}_2, \ldots, \mathtt{x}_k \mapsto \underline{\mathtt{v}}_k\}$$

In particolare, indicheremo con $\varphi = \omega$ uno stato *vuoto*, ovvero tale che $\varphi(\mathtt{x}) = \perp$ qualunque sia `x`. Si noti che, in questa notazione, è essenziale che gli identificatori $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_k$ siano tutti distinti tra loro, mentre non è importante l'ordine in cui vengono elencate le associazioni $\mathtt{x}_i \mapsto \underline{\mathtt{v}}_i$ all'interno delle parentesi graffe.

4.1.1 Espressioni con stato

La dipendenza delle espressioni dallo stato (ovvero dai valori associati ai nomi che compaiono nell'espressione stessa), si riflette, nei sistemi di transizioni, nel fatto che le configurazioni non terminali del sistema stesso sono ora coppie del tipo $\langle \mathtt{E}, \varphi \rangle$. Nella nuova semantica, il significato di una derivazione $\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}$ è che il valore di `E`, date le associazioni in φ, è $\underline{\mathtt{n}}$.

[6] `Ide` è la categoria sintattica a partire dalla quale è possibile generare identificatori.

[7] Esistono tuttavia espressioni il cui valore è indipendente dal valore associato ai nomi che vi compaiono, come ad esempio `x-x`.

$$\langle \mathtt{n}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad (exp_n)$$

$$\frac{\varphi(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \varphi\rangle \to_{exp} \underline{\mathtt{n}}} \qquad (exp_{ide})$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\langle \mathtt{E}+\mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{m}}} \qquad (exp_+)$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\langle \mathtt{E}*\mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{m}}} \qquad (exp_*)$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\langle \mathtt{E}\text{-}\mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{m}}} \qquad (exp_-)$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ div\ \underline{\mathtt{n}}'}{\langle \mathtt{E}\ /\ \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{m}}} \qquad (exp_{div})$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ mod\ \underline{\mathtt{n}}'}{\langle \mathtt{E}\ \%\ \mathtt{E}', \varphi\rangle \to_{exp} \underline{\mathtt{m}}} \qquad (exp_{mod})$$

$$\frac{\langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{n}}}{\langle (\mathtt{E}), \varphi\rangle \to_{exp} \underline{\mathtt{n}}} \qquad (exp_{()})$$

Abbiamo dovuto introdurre una nuova regola corrispondente alla nuova alternativa sintattica per le espressioni: (exp_{ide}) esprime il fatto che il valore dell'espressione costituita dal solo identificatore `x` nello stato φ non è altro che il valore associato ad `x` in tale stato, ovvero $\varphi(\mathtt{x})$.

La semantica di un'espressione è ora parametrica rispetto allo stato ed è quindi definita come:

$$\mathcal{E}[\![\mathtt{E}]\!]\varphi = \underline{\mathtt{k}} \quad \textbf{se e solo se} \quad \langle \mathtt{E}, \varphi\rangle \to_{exp} \underline{\mathtt{k}}.$$

Esempio 4.1 Vediamo come esempio la semantica dell'espressione `(x+8) / y` a partire da uno stato φ in cui al nome `x` è associato il valore $\underline{5}$ ed al nome `y` il valore $\underline{2}$, ovvero da uno stato φ tale che $\varphi(\mathtt{x}) = \underline{5}$ e $\varphi(\mathtt{y}) = \underline{2}$.

$\langle$`(x+8) / y`$, \varphi\rangle$

$\to_{exp}$ { (exp_{div}),
(d1): $\langle$`(x+8)`$, \varphi\rangle \to_{exp} \underline{13}$,
(exp_{ide}): $\langle$`y`$, \varphi\rangle \to_{exp} \underline{2}$
$\underline{2} \neq \underline{0}$, $\underline{13}\ div\ \underline{2} = \underline{6}$ }

$\underline{6}$

(d1):

$\langle$`(x+8)`$, \varphi\rangle$

$\to_{exp}$ { $(exp_{()})$,
(d2): $\langle$`x+8`$, \varphi\rangle \to_{exp} \underline{13}$ }

$\underline{13}$

(d2):

$\langle$x+8, $\varphi\rangle$

$\rightarrow_{exp}$ { (exp_+),
(exp_{ide}): $\langle$x, $\varphi\rangle \rightarrow_{exp} \underline{5}$,
($exp^{:}_{n}$) $\langle 8, \varphi\rangle \rightarrow_{exp} \underline{8}$,
$\underline{13} = \underline{5} + \underline{8}$ }

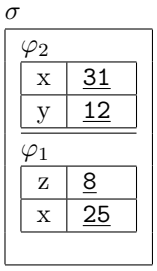
$\underline{13}$

■

4.2 Strutturazione dello stato

Per poter trattare correttamente caratteristiche del linguaggio Java più complesse quali l'annidamento dei blocchi e le chiamate ai metodi, è necessario dare allo stato una strutturazione più sofisticata rispetto a quella introdotta nella Sezione 4.1. Intuitivamente, invece di trattare lo stato come una tabella di associazioni tra identificatori e valori, lo vogliamo trattare come una *pila*, ovvero come una sequenza di tabelle, ciascuna delle quali è un insieme di associazioni tra identificatori e valori. Useremo il termine *frame* per fare riferimento ad una tabella[8]. Il nuovo stato è ora una pila di frame. Nel seguito useremo φ, φ', ecc. per denotare frame e $\sigma, \sigma', ecc.$ per denotare stati, ovvero sequenze di frame.

Si consideri la rappresentazione grafica di uno stato σ costituito da due frame φ_1 e φ_2.



La questione cruciale è che in uno stato possono essere presenti contemporaneamente più associazioni per uno stesso identificatore: nell'esempio, per l'identificatore `x` esistono due associazioni nello stato σ, una nel frame φ_1 e l'altra nel frame φ_2. Ma quale è allora il valore di un identificatore in uno stato? Nell'esempio, quale tra i valori $\underline{25}$ e $\underline{31}$ è il valore di `x` nello stato σ? È il valore che si trova ricercando un'associazione per `x` a partire dal frame più "recente".

Intuitivamente, essendo uno stato una *pila* di frame (una struttura cioè in cui un frame viene sempre "impilato" sull'ultimo creato), il valore di un identificatore è quello che si trova ricercandolo nei frame dello stato dall'alto verso il basso. Nell'esempio, il valore dell'identificatore `x` è $\underline{31}$: il frame φ_2 è stato infatti impilato sul frame φ_1.

Vediamo come si può formalizzare la nuova strutturazione dello stato come pila di frame. Osserviamo innanzitutto che il concetto di frame corrisponde essenzialmente al concetto di stato così come definito nella sezione 4.1: in altre parole un frame è una funzione che associa un valore ad un numero finito di identificatori in `Ide`. Seguendo le notazioni già introdotte, indicheremo con φ(`x`) il valore associato all'identificatore `x` nel frame φ, e rappresenteremo con φ(`x`) $= \perp$ il fatto che il frame φ non associa alcun valore all'identificatore `x`. Infine, indicheremo con ω il frame vuoto, ovvero il frame in cui, per ogni identificatore `x`, ω(`x`) $= \perp$. Nel seguito, indicheremo con Φ l'insieme dei possibili frame.

Uno *stato* è ora una sequenza di frame. Se σ è uno stato e φ un frame, indichiamo con $\varphi.\sigma$ il nuovo stato ottenuto aggiungendo il frame φ alla sequenza σ: nel nuovo stato così ottenuto, φ è il frame *più recente*. Indicando con Ω lo stato vuoto (ovvero la sequenza vuota di frame), l'insieme Σ degli stati può allora

[8] Dunque un frame corrisponde allo stato della sezione 4.1.

essere definito induttivamente come segue:

$$\Sigma = \{\Omega\} \cup \{\varphi.\sigma | \varphi \in \Phi, \sigma \in \Sigma\}.$$

Una formalizzazione dello stato σ dell'esempio precedente è la sequenza:

$$\varphi_2.\varphi_1.\Omega$$

dove φ_1 e φ_2 sono le funzioni:

$$\varphi_1(v) = \begin{cases} \underline{8} & \text{se } v{=}\mathtt{z} \\ \underline{25} & \text{se } v{=}\mathtt{x} \\ \bot & \text{altrimenti} \end{cases} \qquad \varphi_2(v) = \begin{cases} \underline{12} & \text{se } v{=}\mathtt{y} \\ \underline{31} & \text{se } v{=}\mathtt{x} \\ \bot & \text{altrimenti} \end{cases}$$

Dato uno stato σ ed un identificatore $\mathtt{x}$, continuiamo ad indicare con $\sigma(\mathtt{x})$ il valore associato ad $\mathtt{x}$ in σ[9]. Avremo allora:

$$\sigma(\mathtt{x}) = \begin{cases} \bot & \text{se } \sigma = \Omega \\ \varphi(\mathtt{x}) & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) \neq \bot \\ \sigma'(\mathtt{x}) & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) = \bot \end{cases}$$

Consideriamo ancora l'esempio precedente, con $\sigma = \varphi_2.\varphi_1.\Omega$. Abbiamo: $\sigma(\mathtt{x}) = \underline{31}$ poiché $\varphi_2(\mathtt{x}) = \underline{31}$. Inoltre, $\sigma(\mathtt{z}) = \underline{8}$ poiché $\varphi_2(\mathtt{z}) = \bot$ e $\varphi_1(\mathtt{z}) = \underline{8}$.

Osserviamo infine che, con la nuova definizione di stato strutturato, le regole date nel paragrafo 4.1.1 rimangono immutate, con l'unica accortezza che, nelle configurazioni del sistema di transizione $\langle \mathtt{E}, \varphi \rangle$ ivi riportate la seconda componente è uno stato strutturato. Riportiamo come esempio la regola (exp_{ide}):

$$\frac{\sigma(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \sigma \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{ide})$$

dove, secondo le convenzioni appena adottate, abbiamo indicato con σ la seconda componente delle configurazioni, trattandosi di uno stato strutturato.

4.3 Modifica dello stato

Come vedremo in seguito, il significato di un programma è quello di eseguire transizioni di stato, cioè modificare lo stato. A questo proposito, introduciamo una notazione compatta per esprimere le modifiche di stato: l'effetto di queste ultime è di cambiare il valore associato ad uno o più identificatori. Come abbiamo visto nella precedente sezione, un medesimo stato può contenere più associazioni per lo stesso identificatore e dunque bisogna fare attenzione a quali associazioni sono coinvolte nella modifica.

Introduciamo dapprima una notazione per la modifica di un frame, o meglio del valore associato ad uno o più identificatori in un frame. Siano allora φ un frame, $\mathtt{x}$ un identificatore e $\underline{\mathtt{v}}$ un elemento di $\mathtt{Val}$. Denotiamo con $\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ un frame in cui al nome $\mathtt{x}$ è associato $\underline{\mathtt{v}}$, mentre ad ogni altro identificatore $\mathtt{y}$ (diverso da $\mathtt{x}$) è associato lo stesso valore associato ad $\mathtt{y}$ in φ. Ricordando che un frame φ non è altro che una funzione $\varphi : \mathtt{Ide} \mapsto \mathtt{Val}$, quanto appena detto può essere espresso formalmente in questo modo:

$$\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}](y) = \begin{cases} \varphi(y) & \text{se } y \neq \mathtt{x} \\ \underline{\mathtt{v}} & \text{se } y = \mathtt{x} \end{cases}$$

Si noti che $\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ è ancora un frame, ovvero una funzione da $\mathtt{Ide}$ in $\mathtt{Val}$. La precedente notazione può essere estesa per trattare la modifica di un frame in corrispondenza di più identificatori. Siano dunque φ un frame, $\mathtt{x}_1, \ldots \mathtt{x}_k$ una sequenza di k identificatori *distinti* e $\underline{\mathtt{v}}_1, \ldots \underline{\mathtt{v}}_k$ una sequenza di k elementi in $\mathtt{Val}$. La notazione

$$\varphi[{}^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1}, \ldots, {}^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}]$$

[9] Si tratta di un piccolo abuso di notazione, essendo σ una sequenza e non una funzione come φ nella Sezione 4.1.

denota il frame:

$$\varphi[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1},\ldots,^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}](y) = \begin{cases} \underline{\mathtt{v}}_1 & \text{se } y = \mathtt{x}_1 \\ \underline{\mathtt{v}}_2 & \text{se } y = \mathtt{x}_2 \\ \qquad \cdots & \\ \underline{\mathtt{v}}_k & \text{se } y = \mathtt{x}_k \\ \varphi(y) & \text{altrimenti} \end{cases}$$

Come esempio, dato il frame $\varphi = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{2}\}$ e detto φ_1 il frame $\varphi[^{\underline{4}}/_{\mathtt{x}},^{\underline{3}}/_{\mathtt{y}}]$, avremo che $\varphi_1(\mathtt{x}) = \underline{4}$, $\varphi_1(\mathtt{y}) = \underline{3}$ e $\varphi_1(\mathtt{z}) = \perp$ per ogni altro identificatore $\mathtt{z}$.

È importante rimarcare che la condizione che gli identificatori $\mathtt{x}_1, \ldots, \mathtt{x}_k$ siano tutti *distinti* in

$$\varphi[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1},\ldots,^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}]$$

garantisce che quanto ottenuto sia ancora una funzione da `Ide` in `Val`. Laddove fosse necessario rappresentare modifiche successive per lo stesso identificatore, non potremo utilizzare la notazione appena introdotta per modifiche multiple. Così, ad esempio, saremo autorizzati a scrivere $\varphi' = (\varphi[^{\underline{10}}/_{\mathtt{x}}])[^{\underline{20}}/_{\mathtt{x}}]$, ma non $\varphi'' = \varphi[^{\underline{10}}/_{\mathtt{x}},^{\underline{20}}/_{\mathtt{x}}]$. Infatti, mentre dalle definizioni precedenti è chiaro che φ' è una funzione e che $\varphi'(\mathtt{x}) = \underline{20}$, è anche chiaro che φ'' non è una *funzione* da `Ide` in `Val` (quale sarebbe, infatti, l'*unico* valore associato ad $\mathtt{x}$ in φ''?).

Come ulteriore esempio, dato il frame $\varphi = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{2}\}$, il frame $\varphi[^{\underline{4}}/_{\mathtt{x}},^{\underline{3}}/_{\mathtt{z}}]$ è uguale a $\varphi' = \{\mathtt{x} \mapsto \underline{4}, \mathtt{y} \mapsto \underline{2}, \mathtt{z} \mapsto \underline{3}\}$ (verificarlo per esercizio).

Veniamo ora alla modifica di stato, riservando per essa la stessa notazione usata per la modifica di frame: $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ indica la modifica del valore associato ad $\mathtt{x}$ nello stato σ. Intuitivamente, così come il valore associato ad un identificatore $\mathtt{x}$ in uno stato è quello che si determina trovando una associazione per $\mathtt{x}$ a partire dal frame più recente, anche la modifica $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ riguarda l'associazione più recente per $\mathtt{x}$. Detto altrimenti, nello stato $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ viene modificato il frame più recente che contiene una associazione per $\mathtt{x}$. Consideriamo ad esempio uno stato σ del tipo

$$\varphi_1.\varphi_2 \ldots \varphi_i \ldots \varphi_k.\Omega$$

Se φ_i è il primo frame nella sequenza che contiene una associazione per $\mathtt{x}$ (o detto altrimenti $\varphi_1(\mathtt{x}) = \varphi_2(\mathtt{x}) = \ldots = \varphi_{i-1}(\mathtt{x}) = \perp$ e $\varphi_i(\mathtt{x}) \neq \perp$), lo stato $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ non è altro che la sequenza modificata

$$\varphi_1.\varphi_2 \ldots \varphi_i[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] \ldots \varphi_k.\Omega$$

Si noti che, se nessun frame nello stato contiene una associazione per $\mathtt{x}$, la modifica non ha alcun effetto. Nell'esempio, se $\varphi_1(\mathtt{x}) = \varphi_2(\mathtt{x}) = \ldots = \varphi_k(\mathtt{x}) = \perp$, allora $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] = \sigma$.

Una definizione formale di $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ può essere data nel modo seguente.

$$\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] = \begin{cases} \varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}].\sigma' & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) \neq \perp \\ \varphi.\sigma'[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) = \perp \\ \Omega & \text{se } \sigma = \Omega \end{cases}$$

È chiaro che la notazione può essere estesa facilmente per trattare la modifica di una sequenza di identificatori *distinti*, che indicheremo con

$$\sigma[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1},\ldots,^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}].$$

Consideriamo come esempio lo stato $\sigma = \varphi_2.\varphi_1.\Omega$, dove φ_1 e φ_2 sono i frame seguenti:

$$\varphi_1(v) = \begin{cases} \underline{8} & \text{se } v{=}\mathtt{z} \\ \underline{25} & \text{se } v{=}\mathtt{x} \\ \perp & \text{altrimenti} \end{cases} \qquad \varphi_2(v) = \begin{cases} \underline{12} & \text{se } v{=}\mathtt{y} \\ \underline{31} & \text{se } v{=}\mathtt{x} \\ \perp & \text{altrimenti} \end{cases}$$

Lo stato $\sigma[^{\underline{50}}/_{\mathtt{z}},^{\underline{40}}/_{\mathtt{x}}]$ è la sequenza $\varphi_2'.\varphi_1'.\Omega$, con $\varphi_1' = \varphi_1[^{\underline{50}}/_{\mathtt{z}}]$ e $\varphi_2' = \varphi_2[^{\underline{40}}/_{\mathtt{x}}]$. D'altra parte, lo stato $\sigma[^{\underline{10}}/_{\mathtt{w}}] = \sigma$ dal momento che φ_2 e φ_1 non contengono una associazione per $\mathtt{w}$.

Concludiamo questa sezione osservando alcune differenze tra le possibili notazioni introdotte sin qui riguardo a stati e frame. Innanzitutto, è importante osservare che la scrittura $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$ (risp. $\sigma(\mathtt{x}) = \underline{\mathtt{v}}$) stabilisce semplicemente che il valore associato ad $\mathtt{x}$ nel frame φ (risp. nello stato σ) è $\underline{\mathtt{v}}$, mentre è *ignoto* (e non necessariamente *indefinito*) il valore associato ad ogni altro identificatore che non sia $\mathtt{x}$. La scrittura $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$ non equivale a dire che il frame φ contiene una sola associazione (quella per l'identificatore $\mathtt{x}$): non corrisponde cioè a stabilire che $\varphi = \{\mathtt{x} \mapsto \underline{\mathtt{v}}\}$. Infatti, la scrittura $\{\mathtt{x} \mapsto \underline{\mathtt{v}}\}$ indica un frame in cui il valore associato ad $\mathtt{x}$ è $\underline{\mathtt{v}}$, mentre è *indefinito* ($\perp$) il valore associato ad ogni altro identificatore. Inoltre, entrambe le scritture precedenti non equivalgono a $\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$, in quanto quest'ultima sta ad indicare un frame che associa ad $\mathtt{x}$ il valore $\underline{\mathtt{v}}$ e si comporta come φ, qualunque esso sia, in corrispondenza di ogni altro identificatore. Concludiamo infine notando che mentre $\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ rappresenta comunque un frame diverso da φ (con la particolarissima eccezione in cui $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$), $\sigma[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ può essere una sequenza di frame identica a σ.

Esercizi

1. Dire quali sono i valori associati agli identificatori $\mathtt{x}$ ed $\mathtt{y}$ nel frame φ in tutti i casi seguenti:
   (a) $\varphi = \{\mathtt{x} \mapsto \underline{\mathtt{10}}, \mathtt{y} \mapsto \underline{\mathtt{10}}\}$
   (b) $\varphi = \{\mathtt{x} \mapsto \underline{\mathtt{10}}, \mathtt{z} \mapsto \underline{\mathtt{10}}\}$
   (c) $\varphi = \varphi'$
   (d) $\varphi = \varphi'[{}^{\underline{\mathtt{10}}}/_{\mathtt{y}}]$
   (e) $\varphi = \varphi'[{}^{\underline{\mathtt{10}}}/_{\mathtt{y}}, {}^{\underline{\mathtt{20}}}/_{\mathtt{x}}]$
   (f) $\varphi = (\varphi'[{}^{\underline{\mathtt{10}}}/_{\mathtt{y}}, {}^{\underline{\mathtt{20}}}/_{\mathtt{x}}])[{}^{\underline{\mathtt{10}}}/_{\mathtt{x}}]$

2. Siano $\varphi = \{\mathtt{x} \mapsto \underline{\mathtt{10}}, \mathtt{y} \mapsto \underline{\mathtt{10}}\}$ e $\varphi' = \{\mathtt{y} \mapsto \underline{\mathtt{10}}, \mathtt{x} \mapsto \underline{\mathtt{10}}, \mathtt{z} \mapsto \underline{\mathtt{10}}\}$. Mostrare che φ' e $\varphi[{}^{\underline{\mathtt{10}}}/_{\mathtt{z}}]$ denotano lo stesso frame (ovvero la stessa funzione).

3. Mostrare che $(\varphi[{}^{\underline{\mathtt{v}}'}/_{\mathtt{x}}])[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ e $\varphi[{}^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ denotano lo stesso frame, qualunque siano $\varphi, \underline{\mathtt{v}}, \underline{\mathtt{v}}'$ e $\mathtt{x}$.

4. Siano
   $\varphi_1 = \{\mathtt{x} \mapsto \underline{\mathtt{10}}, \mathtt{y} \mapsto \underline{\mathtt{10}}\}$
   e
   $\varphi_2 = \{\mathtt{x} \mapsto \underline{\mathtt{9}}, \mathtt{z} \mapsto \underline{\mathtt{10}}\}$.
   Dire quali sono i valori associati agli identificatori $\mathtt{x}$, $\mathtt{y}$ e $\mathtt{z}$ nello stato σ in tutti i casi seguenti:
   (a) $\sigma = \varphi_1.\varphi_2.\Omega$
   (b) $\sigma = \varphi_2.\varphi_1.\Omega$
   (c) $\sigma = \varphi_1.\varphi_2[{}^{\underline{\mathtt{8}}}/_{\mathtt{x}}].\Omega$
   (d) $\sigma = \varphi_1[{}^{\underline{\mathtt{8}}}/_{\mathtt{x}}].\varphi_2.\Omega$

5 La semantica operazionale del nucleo di Java

A partire da questo paragrafo affronteremo lo studio della semantica in stile operazionale di un frammento di linguaggio di programmazione: il nucleo imperativo di Java. Ci limiteremo a considerare i tipi di dato elementari `int` e `boolean`, le dichiarazioni di variabili limitate ai tipi suddetti ed i costrutti imperativi di base, nelle loro forme più semplici: assegnamento, blocco, condizionale, comando iterativo `while`. Per la sua eccessiva semplicità il frammento da noi considerato non si presenta come un linguaggio di programmazione reale, ma solo come uno strumento per lo studio dei concetti essenziali nella semantica dei linguaggi.

5.1 Le espressioni e la loro semantica

Il primo aspetto da affrontare nello studio di un linguaggio riguarda i meccanismi che esso mette a disposizione per rappresentare e manipolare dati. Nel nostro caso, i tipi di dato presenti nel linguaggio sono solamente valori interi e logici (booleani) ed i costrutti per manipolarli consentono di definire espressioni sia a valori interi che a valori booleani. Questi ultimi sono i valori di verità nell'insieme $\mathbb{B} = \{tt, ff\}$ che verranno rappresentati sintatticamente con i simboli `true` e `false` rispettivamente: il linguaggio mette a disposizione una serie di operatori per definire espressioni (le *proposizioni*) il cui valore è un valore di verità (ad esempio `5>3`) e per combinare tali espressioni. Il significato di tali operatori viene dato formalmente in questo paragrafo. La sintassi delle espressioni è la seguente:

Exp ::= ConstVal | Ide | (Exp) | Exp Op Exp |! Exp
Op ::= +| − | ∗ |/|%|==|! =| < |<=| > |>=|&&|||
ConstVal ::= Num | Bool
Bool ::= true | false
Num ::= ...

Nel seguito, ometteremo la distinzione esplicita tra *valori* e *rappresentazioni* e utilizzeremo le seguenti convenzioni:

- $\mathtt{n}, \mathtt{m}, \mathtt{n}', \mathtt{m}', \ldots$ denotano valori in $\mathbb{N}$, o loro rappresentazioni ;
- $\mathtt{b}, \mathtt{b}', \mathtt{b}_1, \ldots$ denotano valori in $\mathbb{B}$, o loro rappresentazioni;
- $\mathtt{E}, \mathtt{E}', \ldots$ denotano generiche espressioni
- $\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots$ denotano identificatori.

Abbiamo già discusso nel paragrafo 3.2 la semantica delle espressioni a valori naturali. Per quanto concerne le espressioni a valori booleani, osserviamo che gli operatori `&&`, || e `!` corrispondono agli operatori logici $\wedge, \vee$ e $\neg$. Come nel caso delle espressioni a valori naturali, le regole che definiscono la semantica delle espressioni booleane sono guidate dalla sintassi e, in esse, utilizziamo delle premesse che corrispondono ad operazioni "elementari" tra valori di verità che si suppone di saper calcolare.

$$\langle \mathtt{true}, \sigma \rangle \rightarrow_{exp} tt \qquad (exp_{true})$$

$$\langle \mathtt{false}, \sigma \rangle \rightarrow_{exp} ff \qquad (exp_{false})$$

Le due regole precedenti corrispondono allo schema di regola (exp_n) per le costanti naturali.

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \langle \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}' \qquad \mathtt{b}'' = \mathtt{b} \ \vee \ \mathtt{b}'}{\langle \mathtt{E} \ || \ \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}''} \qquad (exp_{or})$$

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \langle \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}' \qquad \mathtt{b}'' = \mathtt{b} \ \wedge \ \mathtt{b}'}{\langle \mathtt{E} \ \&\& \ \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}''} \qquad (exp_{and})$$

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \mathtt{b}' = \neg \mathtt{b}}{\langle \mathtt{!E}, \sigma \rangle \rightarrow_{exp} \mathtt{b}'} \qquad (exp_{not})$$

Queste regole definiscono gli operatori del linguaggio `&&`, `||` e `!` in termini dei corrispondenti operatori logici $\wedge, \vee$ e $\neg$ [10].

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{n} \qquad \langle \mathtt{E}', \sigma\rangle \rightarrow_{exp} \mathtt{n}' \qquad \mathtt{b} = \mathtt{n}\ rel\ \mathtt{n}'}{\langle \mathtt{E}\ \mathtt{rel}\ \mathtt{E}', \sigma\rangle \rightarrow_{exp} \mathtt{b}} \qquad (exp_{rel})$$

Con questa regola abbiamo in realtà espresso tutte le regole che riguardano gli operatori di relazione (confronto) previsti nella sintassi del linguaggio. Il simbolo `rel` sta per uno qualunque dei sei simboli sintattici `==`, `!=`, `<`, `>`, `>=` e `<=` previsti dal linguaggio, mentre il simbolo rel sta per la corrispondente funzione semantica ($=, \neq, \prec, \succ, \succeq, \preceq$ rispettivamente). Dunque (exp_{rel}) esprime contemporaneamente sei regole.

5.2 I costrutti di controllo e la loro semantica

La porzione di linguaggio vista fino ad ora (le espressioni) fornisce i meccanismi di base a nostra disposizione per rappresentare dati e per manipolarli. Un linguaggio che si limitasse a ciò consentirebbe di risolvere problemi la cui soluzione si limita semplicemente al calcolo di espressioni. Tuttavia i problemi che si affrontano nella pratica sono quasi sempre di natura più complessa e la loro soluzione in termini di puro calcolo di espressioni risulterebbe troppo complicata se non impossibile.

In questo paragrafo studiamo un'ulteriore classe di costrutti linguistici, usualmente denominati *comandi*, il cui scopo è essenzialmente quello di consentire la modifica dello stato[11]. Fino ad ora, quest'ultimo è stato introdotto al solo scopo di rappresentare associazioni tra nomi e valori, nomi utilizzabili poi all'interno delle espressioni: in realtà, nei linguaggi imperativi (quelli che stiamo considerando) il ruolo dello stato è ben più rilevante. Un problema di programmazione, in tali linguaggi, si pone proprio in termini della descrizione di uno stato *iniziale* (che rappresenta i *dati* del problema) e di uno stato *finale* (che rappresenta i *risultati* del problema). La soluzione ad un problema siffatto consiste allora nella individuazione di una sequenza di azioni che modificano lo stato iniziale fino a trasformarlo nello stato finale desiderato.

Nel nostro semplice frammento di Java la sintassi dei comandi è data mediante il seguente insieme di produzioni:

Com ::= Ide = Exp;
| Block
| **if** (Exp) Com **else** Com
| **while** (Exp) Com

Block ::= {StatList}
StatList ::= Com | Com StatList

Le configurazioni del sistema di transizioni per i comandi, $\mathtt{S}_{com}$, sono di due tipi: $\langle \mathtt{C}, \sigma\rangle$, dove `C` è un comando e σ uno stato, sono le configurazioni che corrispondono agli stadi iniziali del calcolo (cfr. le configurazioni $\langle \mathtt{E}, \sigma\rangle$ nel sistema $\mathtt{S}_{exp}$), mentre le configurazioni composte da un solo stato σ sono le configurazioni terminali, che corrispondono alla conclusione del calcolo. Riassumendo:

$$\begin{aligned} \Gamma_{com} &= \{\langle \mathtt{C}, \sigma\rangle \mid \mathtt{C} \in \mathtt{Com},\ \ \sigma \in \Sigma\} \cup \\ &\quad\ \{\sigma \mid \sigma \in \Sigma\} \end{aligned}$$

$$\mathtt{T}_{com} = \{\sigma \mid \sigma \in \Sigma\}$$

[10] Rammentiamo che, se b, b' sono elementi di $\mathbb{B}$, $b \wedge b'$ ha valore tt se e solo se b e b' sono entrambi tt, $b \vee b'$ ha valore tt se e solo se almeno uno tra b e b' è tt e $\neg b$ è tt se e solo se b è ff.

[11] Più avanti ci occuperemo anche di introdurre le dichiarazioni, ovvero i costrutti che consentono di definire l'insieme delle associazioni che compongono uno stato.

Le regole che definiscono le transizioni di stato per i comandi sono le seguenti:

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{v}}{\langle \mathtt{x{=}E;}, \sigma\rangle \rightarrow_{com} \sigma[{}^{\mathtt{v}}/_{\mathtt{x}}]} \qquad (com_{=})$$

Il comando di assegnamento è il costrutto del linguaggio che consente di apportare modifiche allo stato. Si noti l'utilizzo della notazione $\sigma[{}^{\mathtt{v}}/_{\mathtt{x}}]$ introdotta nella Sezione 4.2 per rappresentare la modifica di stato.

Tutti gli altri comandi servono per controllare l'uso degli assegnamenti, ovvero delle modifiche di stato, e per questo vengono chiamati *costrutti di controllo*.

Il blocco consente di rappresentare una sequenza di comandi, da eseguire nell'ordine in cui essi compaiono nella sequenza.

$$\frac{\langle \mathtt{Slist}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \{\mathtt{Slist}\}, \sigma\rangle \rightarrow_{com} \sigma'} \qquad (com_{Block})$$

$$\frac{\langle \mathtt{S}, \sigma\rangle \rightarrow_{com} \sigma'' \qquad \langle \mathtt{Slist}, \sigma''\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{S\ \ Slist}, \sigma\rangle \rightarrow_{com} \sigma'} \qquad (com_{Stat-list})$$

La regola $(com_{Stat-list})$ esprime il fatto che, in una sequenza del tipo $\mathtt{S}_1\mathtt{S}_2 \ldots \mathtt{S}_k$, la sottosequenza $\mathtt{S}_2 \ldots \mathtt{S}_k$ inizia ad operare nello stato che risulta dal calcolo corrispondente al primo comando $\mathtt{S}_1$ dell'intera sequenza.

Esempio 5.1 Consideriamo la lista di comandi:

```
x = 2; y = 3; x = x-1;
```

e lo stato $\sigma_0 = \{\mathtt{x} \mapsto \underline{0}, \mathtt{y} \mapsto \underline{0}\}.\Omega$. Abbiamo dunque la derivazione:

$\langle \mathtt{x} = 2; \mathtt{y} = 3; \mathtt{x} = \mathtt{x} - 1;, \sigma_0\rangle$

$\rightarrow_{com}$ { $(com_{Stat-list})$,
(d1): $\langle \mathtt{x} = 2;, \sigma_0\rangle \rightarrow_{com} \sigma_0[{}^{\underline{2}}/_{\mathtt{x}}]$,
$\sigma_1 = \sigma_0[{}^{\underline{2}}/_{\mathtt{x}}]$,
(d2): $\langle \mathtt{y} = 3; \mathtt{x} = \mathtt{x} - 1;, \sigma_1\rangle \rightarrow_{com} \sigma_1[{}^{\underline{1}}/_{\mathtt{x}}, {}^{\underline{3}}/_{\mathtt{y}}]$
$\sigma_1[{}^{\underline{1}}/_{\mathtt{x}}, {}^{\underline{3}}/_{\mathtt{y}}] = \sigma_0[{}^{\underline{1}}/_{\mathtt{x}}, {}^{\underline{3}}/_{\mathtt{y}}]$}

$\sigma_0[{}^{\underline{1}}/_{\mathtt{x}}, {}^{\underline{3}}/_{\mathtt{y}}] = \{\mathtt{x} \mapsto \underline{1}, \mathtt{y} \mapsto \underline{3}\}.\Omega$

dove (d1) è la derivazione:

(d1):
$\langle \mathtt{x} = 2;, \sigma_0\rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\mathcal{E}[\![2]\!]\sigma_0 = \underline{2}$}

$\sigma_0[{}^{\underline{2}}/_{\mathtt{x}}]$

e, detto σ_1 lo stato $\sigma_0[{}^{\underline{2}}/_{\mathtt{x}}]$, (d2) è la derivazione:

(d2):

$\langle \mathtt{y} = \mathtt{3}; \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma_1 \rangle$

$\rightarrow_{com}$ { $(com_{Stat-list})$,
(d3): $\langle \mathtt{y} = \mathtt{3};, \sigma_1 \rangle \rightarrow_{com} \sigma_1[\underline{3}/\mathtt{y}]$,
$\sigma' = \sigma_1[\underline{3}/\mathtt{y}]$,
(d4): $\langle \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma' \rangle \rightarrow_{com} \sigma'[\underline{1}/\mathtt{x}]$,
$\sigma'[\underline{1}/\mathtt{x}] = \sigma_0[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}]$}

$\sigma_0[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}]$

Infine, le sotto-derivazioni (d3) e (d4) sono le seguenti:

(d3):

$\langle \mathtt{y} = \mathtt{3};, \sigma_0[\underline{2}/\mathtt{x}] \rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\mathcal{E}[\![\mathtt{3}]\!]\sigma_0[\underline{2}/\mathtt{x}] = \underline{3}$}

$\sigma_0[\underline{2}/\mathtt{x},\underline{3}/\mathtt{y}]$

(d4):

$\langle \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma_0[\underline{2}/\mathtt{x},\underline{3}/\mathtt{y}] \rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\mathcal{E}[\![\mathtt{x} - \mathtt{1}]\!]\sigma_0[\underline{2}/\mathtt{x},\underline{3}/\mathtt{y}] \rangle = \underline{1}$}

$\sigma_0[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}]$

■

Trattandosi del primo esempio di derivazione in $\rightarrow_{com}$ abbiamo esplicitato tutte le sottoderivazioni e le loro giustificazioni. Si noti che le uguaglianze del tipo $\sigma' = \sigma[\ldots]$ che compaiono nei vari passi, non corrispondono a premesse di regole, ma semplicemente all'introduzione di nomi di comodo per gli stati intermedi del calcolo.

Anche le sottoderivazioni che corrispondono a singoli comandi di assegnamento sono sufficientemente semplici da poter essere esplicitate direttamente nelle giustificazioni. Adotteremo di qui in poi questa convenzione, messa in luce nell'esempio seguente.

Esempio 5.2 Consideriamo di nuovo il comando dell'esempio precedente.

$\langle \mathtt{x} = \mathtt{2}; \mathtt{y} = \mathtt{3}; \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma_0 \rangle$

$\rightarrow_{com}$ { $(com_{Stat-list})$,
$(com_{=})$: $\mathcal{E}[\![\mathtt{2}]\!]\sigma_0 = \underline{2}$,
$\langle \mathtt{x} = \mathtt{2};, \sigma_0 \rangle \rightarrow_{com} \sigma_0[\underline{2}/\mathtt{x}]$,
$\sigma_1 = \sigma_0[\underline{2}/\mathtt{x}]$,
(d1): $\langle \mathtt{y} = \mathtt{3}; \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma_1 \rangle \rightarrow_{com} \sigma_1[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}]$
$\sigma_1[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}] = \sigma_0[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}]$}

$\sigma_0[\underline{1}/\mathtt{x},\underline{3}/\mathtt{y}] = \{\mathtt{x} \mapsto \underline{1}, \mathtt{y} \mapsto \underline{3}\}.\Omega$

dove, detto σ_1 lo stato $\sigma_0[\underline{2}/\mathtt{x}]$, (d1) è la sottoderivazione:

(d1):

$\langle \mathtt{y} = \mathtt{3}; \mathtt{x} = \mathtt{x} - \mathtt{1};, \sigma_1 \rangle$

$\rightarrow_{com}$ $\{$ $(com_{Stat-list})$,
$(com_{=}) : \mathcal{E}[\![\mathtt{3}]\!]\sigma_1 = \underline{3}$,
$\langle \mathtt{y = 3;}, \sigma_1\rangle \rightarrow_{com} \sigma_1[^{\underline{3}}/_{\mathtt{y}}]$,
$\sigma' = \sigma_1[^{\underline{3}}/_{\mathtt{y}}]$,
$(com_{=}) : \mathcal{E}[\![\mathtt{x - 1}]\!]\sigma' = \underline{1}$, $\langle \mathtt{x = x - 1;}, \sigma'\rangle \rightarrow_{com} \sigma'[^{\underline{1}}/_{\mathtt{x}}]$
$\sigma'[^{\underline{1}}/_{\mathtt{x}}] = \sigma_0[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}]\}$

$\sigma_0[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}]$

■

Veniamo infine alle regole per il comando condizionale e per il comando iterativo.

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} tt \qquad \langle \mathtt{C1}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{if-tt})$$

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} ff \qquad \langle \mathtt{C2}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{if-ff})$$

Il comando condizionale consente di scegliere come proseguire il calcolo a seconda del verificarsi o meno di una condizione, rappresentata dall'espressione logica `E`.

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} ff}{\langle \mathtt{while\ (E)\ C}, \sigma\rangle \rightarrow_{com} \sigma} \quad (com_{while-ff})$$

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} tt \qquad \langle \mathtt{C}, \sigma\rangle \rightarrow_{com} \sigma'' \qquad \langle \mathtt{while\ (E)\ C}, \sigma''\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{while\ (E)\ C}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{while-tt})$$

Il comando di iterazione `while (E) C` consente di ripetere l'esecuzione di `C` (detto *corpo*) finché la condizione `E` (detta *guardia*) è verificata. Ciò si riflette in due regole: la prima che identifica la semantica del comando `while` con quella del comando vuoto nel caso in cui la guardia sia falsa; la seconda esprime invece il fatto che, quando la guardia è vera, l'intero comando viene eseguito di nuovo dopo aver eseguito una volta il corpo. Il lettore attento avrà notato che la regola $(com_{while-tt})$ ha come premessa una transizione che riguarda il comando stesso. Non si tratta, tuttavia, di una definizione circolare in quanto la componente stato della configurazione è diversa.

Esempio 5.3 Vediamo come esempio la semantica del comando `while (x>0) x=x-1;` a partire dallo stato $\sigma = \{x \mapsto \underline{2}\}.\Omega$.

$\langle$`while (x>0) x=x-1;`, $\{\mathtt{x} \mapsto \underline{2}\}.\Omega\rangle$

$\rightarrow_{com}$ $\{(com_{while-tt}) :$
$\mathcal{E}[\![\mathtt{x > 0}]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = tt$,
$(com_{=}) : \mathcal{E}[\![\mathtt{x-1}]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = \underline{1}$
$\langle \mathtt{x=x-1;}, \{\mathtt{x} \mapsto \underline{2}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{1}\}.\Omega$,
(d1): $\langle$`while (x>0) x=x-1;`, $\{\mathtt{x} \mapsto \underline{1}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega$ $\}$

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

(d1):

$\langle$`while (x>0) x=x-1;`, $\{\mathtt{x} \mapsto \underline{1}\}.\Omega\rangle$

$\rightarrow_{com}$ { $(com_{while-tt})$:
$\mathcal{E}[\![\mathtt{x} > 0]\!]\{\mathtt{x} \mapsto \underline{1}\}.\Omega = tt$,
$(com_{=})$: $\mathcal{E}[\![\mathtt{x-1}]\!]\{\mathtt{x} \mapsto \underline{1}\}.\Omega = \underline{0}$,
$\langle$`x=x-1;`, $\{\mathtt{x} \mapsto \underline{1}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega$,
(d2): $\langle$`while (x>0) x=x-1;`, $\{\mathtt{x} \mapsto \underline{0}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega$ }

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

(d2):

$\langle$`while (x>0) x=x-1;`, $\{\mathtt{x} \mapsto \underline{0}\}.\Omega\rangle$

$\rightarrow_{com}$ { $(com_{while-ff})$:
$\mathcal{E}[\![\mathtt{x} > 0]\!]\{\mathtt{x} \mapsto \underline{0}\}.\Omega = ff$ }

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

■

Si noti che anche nel caso del sistema $\mathtt{S}_{com}$ esistono configurazioni *bloccate*, alle quali cioè non è possibile applicare alcuna transizione. Un esempio di configurazione siffatta è la configurazione $\langle \mathtt{x} = 5 - 8;, \sigma\rangle$: l'unica regola potenzialmente applicabile è chiaramente $(com_{=})$, che però non può essere applicata in quanto non vi è modo di soddisfare la sua premessa (non esiste infatti alcun $\underline{\mathtt{v}} \in \mathbb{N}$ per cui $\mathcal{E}[\![5 - 8]\!]\sigma = \underline{\mathtt{v}}$). Le configurazioni bloccate di questo tipo corrispondono a situazioni di errore. Si noti come, in generale, il fatto che una configurazione sia bloccata o meno dipende dallo stato: ad esempio, la configurazione $\langle \mathtt{x} \ / \ \mathtt{y}, \sigma\rangle$ è bloccata se, nello stato σ, il valore associato ad `y` è $\underline{0}$, non lo è altrimenti. Errori di questo tipo sono detti *dinamici*, in quanto rilevabili solo al momento dell'esecuzione. Un'altra categoria di errori dinamici corrisponde a configurazioni che danno origine ad una sequenza *infinita* di passi di derivazione. Si consideri ad esempio la configurazione $\langle$`while` $(\mathtt{x} > 0)\ \mathtt{x} = \mathtt{x} + 1;, \{\mathtt{x} \mapsto \underline{2}\}.\Omega\rangle$. L'unica regola del sistema $\mathtt{S}_{com}$ potenzialmente applicabile è chiaramente $(com_{while-tt})$, dal momento che $\mathcal{E}[\![\mathtt{x} > 0]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = tt$. L'applicazione della regola $(com_{while-tt})$ richiede però, tra le premesse, di determinare una transizione del tipo $\langle$`while` $(\mathtt{x} > 0)\ \mathtt{x} = \mathtt{x} + 1;, \{\mathtt{x} \mapsto \underline{3}\}.\Omega\rangle \rightarrow_{com} \sigma$, per qualche σ. Anche in questo caso l'unica regola potenzialmente applicabile è $(com_{while-tt})$ (essendo $\mathcal{E}[\![\mathtt{x} > 0]\!]\{\mathtt{x} \mapsto \underline{3}\}.\Omega = tt$), ed ancora una delle sue premesse corrisponde a determinare una transizione del tipo $\langle$`while` $(\mathtt{x} > 0)\ \mathtt{x} = \mathtt{x} + 1;, \{\mathtt{x} \mapsto \underline{4}\}.\Omega\rangle \rightarrow_{com} \sigma$. È facile convincersi che questo procedimento non ha mai termine (dal momento che nessuno degli stati intermedi del calcolo corrisponde mai ad una situazione in cui la guardia del comando ha valore ff). Nella nostra semantica operazionale le configurazioni bloccate corrispondenti ai due tipi di errori dinamici appena visti non sono distinguibili, anche se esse corrispondono a situazioni diverse dal punto di vista del calcolo. Altri approcci alla semantica operazionale dei linguaggi consentono invece di distinguere vari tipi di configurazioni bloccate.

Esercizi

1. Valutare il blocco

   {`x=y;y=x;`}

   a partire dallo stato σ nei seguenti casi:

   (a) $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{10}\}.\Omega$
   (b) $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{5}\}.\Omega$

2. Valutare il blocco

   {`z=x; x=y; y=z;`}

   a partire dallo stato $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{10}, z \mapsto 0\}.\Omega$

3. Valutare il blocco

   ```
   {z= x/y; w=x%y;
   if (w==0) z=0; else z=z+1;}
   ```

   a partire dallo stato σ nei seguenti casi:

   (a) $\sigma = \{\mathtt{x} \mapsto \underline{15}, \mathtt{y} \mapsto \underline{10}, \mathtt{z} \mapsto \underline{10}, \mathtt{w} \mapsto \underline{100}\}.\Omega$

   (b) $\sigma = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{2}, \mathtt{z} \mapsto \underline{10}, \mathtt{w} \mapsto \underline{100}\}.\Omega$

4. Valutare il blocco

   ```
   {
   z=1;
   while (x != 0) {z=z*x; x=x-1;}
   }
   ```

   a partire da uno stato σ nei seguenti casi:

   (a) $\sigma(\mathtt{x}) = \underline{0}$ e $\sigma(\mathtt{z}) \neq \bot$

   (b) $\sigma(\mathtt{x}) = \underline{4}$ e $\sigma(\mathtt{z}) \neq \bot$

5. Valutare il blocco

   ```
   {
   z = 0;
   while (y<=x) {z = z+1; x = x-y;}
   }
   ```

   a partire da uno stato σ nei seguenti casi:

   (a) $\sigma(\mathtt{x}) = \underline{17}$, $\sigma(\mathtt{y}) = \underline{5}$ e $\sigma(\mathtt{z}) \neq \bot$

   (b) $\sigma(\mathtt{x}) = \underline{5}$, $\sigma(\mathtt{y}) = \underline{8}$ e $\sigma(\mathtt{z}) \neq \bot$

5.3 Dimostrazioni di equivalenza attraverso la semantica operazionale

La semantica operazionale può essere utilizzata per dimostrare proprietà interessanti, come ad esempio l'equivalenza tra espressioni e/o comandi. Informalmente, diciamo che due comandi (espressioni) sono equivalenti se hanno lo stesso comportamento in qualunque contesto, ovvero a partire dallo stesso stato iniziale. Più formalmente ciò può essere espresso come segue.

Definizione 5.4 Due comandi $\mathtt{C}$ e $\mathtt{C}'$ sono *equivalenti* se, per ogni stato σ, vale una delle seguenti condizioni:

a) né $\langle \mathtt{C}, \sigma\rangle$ né $\langle \mathtt{C}', \sigma\rangle$ portano in una configurazione terminale;

b) $\langle \mathtt{C}, \sigma\rangle$ e $\langle \mathtt{C}', \sigma\rangle$ portano nella stessa configurazione terminale. □

Consideriamo ad esempio i due comandi seguenti:

```
C1 :  if (true) x=5; else x=6;
C2 :  x=5;
```

e dimostriamone l'equivalenza. Sia dunque σ un generico stato e valutiamo $\langle \mathtt{C}_1, \sigma\rangle$ e $\langle \mathtt{C}_2, \sigma\rangle$.

$$
\begin{array}{ll}
 & \langle \mathtt{C}_1, \sigma\rangle \\
\rightarrow_{com} & \quad \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{true}]\!]\sigma = tt, \\
 & \quad\ \ (com_{=}) : \mathcal{E}[\![\mathtt{5}]\!]\sigma = \underline{5},\ \langle \mathtt{x = 5;}, \sigma\rangle \rightarrow_{com} \sigma[^{\underline{5}}/_{\mathtt{x}}]\ \} \\
 & \sigma[^{\underline{5}}/_{\mathtt{x}}]
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathtt{C}_2, \sigma\rangle \\
\rightarrow_{com} & \quad \{\ (com_{=}) : \mathcal{E}[\![\mathtt{5}]\!]\sigma = \underline{5}\ \} \\
 & \sigma[^{\underline{5}}/_{\mathtt{x}}]
\end{array}
$$

In questo caso, nelle due derivazioni non abbiamo fatto alcuna ipotesi sullo stato iniziale σ ed abbiamo ottenuto lo stesso stato finale: i due comandi, quindi, a partire da un generico stato portano nella stessa configurazione terminale.

Come ulteriore esempio, consideriamo i due comandi seguenti:

```
C1 :  if (x>2) y=1; else y=2;
C2 :  if (x<=2) y=2; else y=1;
```

La dimostrazione di equivalenza avviene ragionando *per casi*: consideriamo dapprima uno stato σ tale che $\mathcal{E}[\![x > 2]\!]\sigma = tt$. Abbiamo allora le due seguenti derivazioni:

$$
\begin{array}{ll}
 & \langle \mathtt{C}_1, \sigma\rangle \\
\rightarrow_{com} & \quad \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{x > 2}]\!]\sigma = tt, \\
 & \quad\ \ (com_{=}) : \mathcal{E}[\![\mathtt{1}]\!]\sigma = \underline{1},\ \langle \mathtt{y = 1;}, \sigma\rangle \rightarrow_{com} \sigma[^{\underline{1}}/_{\mathtt{y}}]\ \} \\
 & \sigma[^{\underline{1}}/_{\mathtt{y}}]
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathtt{C}_2, \sigma\rangle \\
\rightarrow_{com} & \quad \{\ (com_{if-ff}),\ \mathcal{E}[\![\mathtt{x<=2}]\!]\sigma = ff, \\
 & \quad\ \ (com_{=}) : \mathcal{E}[\![\mathtt{1}]\!]\sigma = \underline{1},\ \langle \mathtt{y = 1;}, \sigma\rangle \rightarrow_{com} \sigma[^{\underline{1}}/_{\mathtt{y}}]\ \} \\
 & \sigma[^{\underline{1}}/_{\mathtt{y}}]
\end{array}
$$

Abbiamo così concluso che i due comandi portano nello stesso stato finale, a partire da un generico stato iniziale in cui $\mathcal{E}[\![\mathtt{x} > \mathtt{2}]\!]\sigma = tt$. Ciò non conclude tuttavia la dimostrazione, che deve considerare anche tutti gli altri stati, ovvero quelli in cui $\mathcal{E}[\![\mathtt{x} > \mathtt{2}]\!]\sigma = ff$. Sia allora σ un generico stato in cui $\mathcal{E}[\![\mathtt{x} > \mathtt{2}]\!]\sigma = ff$. Abbiamo:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{if-ff}),\ \mathcal{E}[\![\mathtt{x} > \mathtt{2}]\!]\sigma = ff, \\ & \quad (com_{=}) : \mathcal{E}[\![\mathtt{2}]\!]\sigma = \underline{2},\ \langle \mathtt{y} = \mathtt{2};, \sigma \rangle \rightarrow_{com} \sigma[^{\underline{2}}/_{\mathtt{y}}]\ \} \\ & \sigma[^{\underline{2}}/_{\mathtt{y}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{x<=2}]\!]\sigma = tt, \\ & \quad (com_{=}) : \mathcal{E}[\![\mathtt{2}]\!]\sigma = \underline{2},\ \langle \mathtt{y} = \mathtt{2};, \sigma \rangle \rightarrow_{com} \sigma[^{\underline{2}}/_{\mathtt{y}}]\ \} \\ & \sigma[^{\underline{2}}/_{\mathtt{y}}] \end{array}$$

Un altro tipo di equivalenza, più debole, è quella che si limita a considerare i due costrutti (comandi o espressioni) in un particolare stato. Consideriamo ad esempio le due espressioni

$\mathtt{E}_1$ =`(x+8) / y` $\qquad$ $\mathtt{E}_2$ =`((x*y)+y) / 2`

e dimostriamone l'equivalenza rispetto ad uno stato σ tale che $\sigma(\mathtt{x}) = \underline{5}$ e $\sigma(\mathtt{y}) = \underline{2}$. Nell'ultimo esempio del paragrafo 4.1 abbiamo visto che:

$\langle \mathtt{E}_1, \sigma \rangle \rightarrow_{exp} \underline{6}$ $\qquad$ (i)

Lasciamo per esercizio al lettore la derivazione

$\langle \mathtt{E}_2, \sigma \rangle \rightarrow_{exp} \underline{6}$

che, insieme con (i), fornisce l'equivalenza di $\mathtt{E}_1$ ed $\mathtt{E}_2$ in σ. Si noti che gli stati σ tali che $\sigma(\mathtt{x}) = \underline{5}$ e $\sigma(\mathtt{y}) = \underline{2}$ sono infiniti, ma questo non basta per dimostrare l'equivalenza di $\mathtt{E}_1$ ed $\mathtt{E}_2$ in tutti i possibili stati. Infatti, in uno stato σ' tale che $\sigma'(\mathtt{x}) = \underline{1}$ e $\sigma'(\mathtt{y}) = \underline{1}$ le due espressioni danno un risultato diverso (verificarlo per esercizio).

Vediamo un ultimo esempio, in cui è ancora necessario un ragionamento *per casi*. Consideriamo i due comandi:

```
C1 :  if (E) x=true; else x=false;
C2 :  x=E;
```

dove `E` è una qualsiasi espressione booleana. Consideriamo un generico stato σ e scindiamo la dimostrazione in tre casi.

(i) $\mathcal{E}[\![\mathtt{E}]\!]\sigma$ non è definita. In questo caso entrambi i comandi non portano in alcuna configurazione terminale.

(ii) $\mathcal{E}[\![\mathtt{E}]\!]\sigma = tt$. In questo caso abbiamo le due derivazioni:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = tt, \\ & \quad (com_{=}) \colon \mathcal{E}[\![\mathtt{true}]\!]\sigma = tt,\ \langle \mathtt{x} = \mathtt{true};, \sigma \rangle \rightarrow_{com} \sigma[^{tt}/_{\mathtt{x}}]\ \} \\ & \sigma[^{tt}/_{\mathtt{x}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{=}) \colon \mathcal{E}[\![\mathtt{E}]\!]\sigma = tt,\ \langle \mathtt{x} = \mathtt{E};, \sigma \rangle \rightarrow_{com} \sigma[^{tt}/_{\mathtt{x}}]\ \} \\ & \sigma[^{tt}/_{\mathtt{x}}] \end{array}$$

(iii) $\mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff}$. In questo caso abbiamo le due derivazioni:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{if-\mathit{ff}}),\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff}, \\ & \quad (com_{=}):\ \mathcal{E}[\![\mathtt{false}]\!]\sigma = \mathit{ff},\ \langle \mathtt{x} = \mathtt{false};, \sigma \rangle \rightarrow_{com} \sigma[^{\mathit{ff}}/_{\mathtt{x}}]\ \} \\ & \sigma[^{\mathit{ff}}/_{\mathtt{x}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma \rangle \\ \rightarrow_{com} & \{\ (com_{=}):\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff},\ \langle \mathtt{x} = \mathtt{E};, \sigma \rangle \rightarrow_{com} \sigma[^{\mathit{ff}}/_{\mathtt{x}}]\ \} \\ & \sigma[^{\mathit{ff}}/_{\mathtt{x}}] \end{array}$$

Poiché i casi (i), (ii) e (iii) esauriscono tutti i possibili, concludiamo che i due comandi sono equivalenti.

Esercizi

1. Verificare l'equivalenza dei due comandi:

   ```
   C1 :  if (true) x=5; else x=3;
   C2 :  if (false) x=3; else x=5;
   ```

2. Verificare l'equivalenza dei due comandi:

   ```
   C1 :  if (x==y) x=x-y; else x=y-y;
   C2 :  x=0;
   ```

3. Dati i due comandi:

   ```
   C1 :  while (x>2) x=x-1;
   C2 :  x=x-1; while (x>2) x=x-1;
   ```

   determinare uno stato in cui sono equivalenti ed uno in cui non lo sono.

4. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari e $\mathtt{C}$, $\mathtt{C}'$ i comandi:

   $\mathtt{C}$: $\mathtt{C}_2\ \mathtt{C}_3$
   $\mathtt{C}'$: $\mathtt{C}_1\ \mathtt{C}_2$

   Dimostrare l'equivalenza di $\mathtt{C}'\ \mathtt{C}_3$ e $\mathtt{C}_1\ \mathtt{C}$

5. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari. Verificare l'equivalenza dei due comandi:

   $\mathtt{C}$: `if` (E) $\{\mathtt{C}_1\ \mathtt{C}_3\}$ `else` $\{\mathtt{C}_2\ \mathtt{C}_3\}$
   $\mathtt{C}'$: `if` (E) $\mathtt{C}_1$ `else` $\mathtt{C}_2\ \mathtt{C}_3$

6. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari. Dare una condizione sufficiente affinché i comandi

   $\mathtt{C}$: $\mathtt{C}_1$ `if` (E) $\mathtt{C}_2$ `else` $\mathtt{C}_3$
   $\mathtt{C}'$: `if` (E) $\{\mathtt{C}_1\ \mathtt{C}_2\}$ `else` $\{\mathtt{C}_1\ \mathtt{C}_3\}$

   siano equivalenti nello stato σ.

5.4 Le dichiarazioni e la loro semantica

Nelle sezioni precedenti abbiamo introdotto i concetti di frame e stato per modellare associazioni tra nomi e valori. Il costrutto linguistico che consente di introdurre i nomi utilizzati in un frammento di programma è quello delle *dichiarazioni*. In questo paragrafo ci occupiamo delle dichiarazioni di *variabili*, che corrispondono ad associazioni modificabili nello stato: la modifica di tali associazioni avviene, come visto nei paragrafi precedenti, mediante il comando di assegnamento. Non ci occuperemo invece delle dichiarazioni di *costanti*, corrispondenti ad associazioni non modificabili. Inoltre, ogni nome viene dichiarato con il suo *tipo*, vale a dire l'insieme dei valori che può essere associato a quel nome. Nel caso del nostro semplice linguaggio ci limitiamo, per il momento, ai tipi `int` e `boolean`, che corrispondono rispettivamente all'insieme dei numeri interi e all'insieme dei valori di verità.

La sintassi di una dichiarazione è data dalle seguenti produzioni:

Decl ::=| Type Ide;
| Type Ide = Exp;

Type ::= int | boolean

Ogni identificatore di variabile, prima di essere utilizzato in una espressione o in un comando, deve essere dichiarato. Una dichiarazione può essere fatta in un qualunque punto di una sequenza all'interno di un blocco. Dal punto di vista sintattico, ciò comporta la seguente modifica delle produzioni date nel paragrafo 5.2.

Block ::= {StatList }

StatList ::= Stat | Stat StatList

Stat ::= Com | Decl

L'utilizzo di un nome all'interno di un assegnamento o di una espressione è lecito se tale nome è stato prima dichiarato.
Vediamo subito alcuni esempi di blocchi.

```
{
    int x;
    x = 10;
    int y;
    y = x + 5;
}
```

Come si vede l'uso di ciascuna variabile all'interno di un comando di assegnamento o di una espressione, segue la sua dichiarazione. Non è invece lecita una sequenza del tipo

```
{
    int x;
    x = 10;
    y = x + 5;
    int y;
}
```

in cui la dichiarazione della variabile `y` segue il suo utilizzo. Vediamo cosa succede nel caso di blocchi annidati.

```
{
    int x;
    x = 10;
        {
            int y;
            y = x + 5;
        }
    ...
}
```

Come si vede nel blocco più interno viene dichiarato l'identificatore `y` e viene usato anche l'identificatore `x`, quest'ultimo dichiarato nel blocco più esterno. Questa situazione è lecita, poiché un nome dichiarato in un blocco è *visibile* all'interno dei blocchi in esso annidati[12]. All'uscita di un blocco, tutti i nomi in esso dichiarati cessano di esistere: questi concetti intuitivi che poco si prestano ad essere spiegati informalmente, troveranno una sistematizzazione chiara e concisa non appena daremo le nuove regole semantiche per le dichiarazioni e per i blocchi.

Nel seguito, dato un blocco, chiameremo variabili *locali* al blocco tutte le variabili dichiarate all'interno del blocco medesimo e chiameremo variabili *globali* al blocco tutte le variabili visibili nel blocco ma non dichiarate in esso.

Prima di formalizzare i concetti sopra esposti, vediamo qualche altro esempio di annidamento.

```
{
    int x;
    x = 10;
        {
            int y;
            y = x + 5;
        }
    x = y+1;
}
```

L'assegnamento `x=y+1` è errato, in quanto il nome `y` non è locale né globale al blocco in cui si trova l'assegnamento stesso. Si noti che, nel blocco più interno, una variabile di nome `y` è stata dichiarata, ma ha cessato di esistere con il termine del blocco annidato.

```
{
    int x;
    x = 10;
        {
            int y;
            y = z + 5;
        }
    int z;
    z = 20;
}
```

L'assegnamento `y = z + 5` è errato in quanto il nome `z` è dichiarato nel blocco più esterno, ma in una posizione successiva al blocco più interno in cui viene utilizzato.

Veniamo alla formalizzazione delle dichiarazioni mediante la semantica operazionale. Introduciamo dapprima un sistema di transizioni per le dichiarazioni, $\mathtt{S}_{dec}$, il cui scopo è di aggiungere nel frame corrente una associazione per ogni nuova variabile dichiarata. Le configurazioni non terminali di $\mathtt{S}_{dec}$ sono del tipo $\langle \mathtt{D}, \sigma \rangle$, dove `D` è un dichiarazione e σ uno stato, mentre le configurazioni terminali sono semplicemente stati.

Osserviamo innanzitutto che una dichiarazione di variabile fissa il tipo dei valori che la variabile potrà

[12] Purché non vi siano dichiarazioni annidate che utilizzano lo stesso nome (si veda l'esempio alla fine di questa sezione).

assumere in seguito: questo è il motivo della presenza del tipo nella dichiarazione. Nonostante sia necessaria, noi non utilizzeremo in questa definizione semantica l'informazione relativa al tipo dei valori[13].

Inoltre, per ciascun tipo elementare `T` (nel nostro caso `int` o `boolean`) una variabile può essere dichiarata senza inizializzarne il valore (ad esempio `int x;`) oppure inizializzandone il valore a quello di una espressione (ad esempio `boolean vero=true;`). Nel primo caso, il valore associato alla variabile è imprecisato: per evidenziare il fatto che un tale valore non fa parte della definizione del linguaggio, utilizziamo il valore *simbolico* ϖ.

$$\frac{\sigma = \varphi.\sigma'' \qquad \sigma' = \varphi[^{\varpi}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x;}, \sigma\rangle \rightarrow_{dec} \sigma'} \qquad (dec_{var-1})$$

$$\frac{\sigma = \varphi.\sigma'' \qquad \langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \varphi[^{\mathtt{v}}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x = E;}, \sigma\rangle \rightarrow_{dec} \sigma'} \qquad (dec_{var-2})$$

Le dichiarazioni, come visto negli esempi e come formalizzato nella sintassi modificata data in precedenza, vengono introdotte nei blocchi. Dunque, la semantica data per questi ultimi con le regole (com_{Block}) e ($com_{Stat-List}$), va modificata opportunamente. Intuitivamente, ogni blocco corrisponde alla introduzione di un nuovo frame nello stato: tale frame è destinato a contenere le associazioni per le variabili dichiarate in quel blocco e cessa di esistere una volta terminata l'esecuzione del blocco. Le liste di dichiarazioni e comandi (elementi di `Stat_list`) vengono trattate attraverso un opportuno sottosistema, le cui transizioni sono date di seguito.

$$\frac{\langle \mathtt{Slist}, \omega.\sigma\rangle \rightarrow_{slist} \varphi.\sigma'}{\langle \{\mathtt{Slist}\}, \sigma\rangle \rightarrow_{com} \sigma'} \qquad (com_{Block})$$

$$\frac{\mathtt{S} \in \mathtt{Decl} \qquad \langle \mathtt{S}, \sigma\rangle \rightarrow_{dec} \sigma'}{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'} \qquad (slist_{dec})$$

$$\frac{\mathtt{S} \in \mathtt{Com} \qquad \langle \mathtt{S}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'} \qquad (slist_{com})$$

$$\frac{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'' \qquad \langle \mathtt{Slist}, \sigma''\rangle \rightarrow_{slist} \sigma'}{\langle \mathtt{S\ \ Slist}, \sigma\rangle \rightarrow_{slist} \sigma'} \qquad (slist_{list})$$

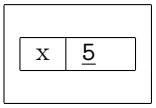
La regola (com_{Block}) formalizza il fatto che un blocco viene eseguito in uno stato nel quale è stato impilato un nuovo frame (inizialmente vuoto): tale frame è quello destinato a contenere le dichiarazioni locali al blocco. La conclusione della regola, inoltre, formalizza il fatto che, nello stato risultante dall'esecuzione del blocco, i nomi locali cessano di esistere mentre permangono le eventuali modifiche apportate a nomi globali al blocco.

Vediamo un esempio di valutazione di un blocco, con l'ausilio della rappresentazione grafica dello stato e dei frame in gioco.

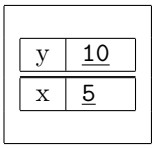
[13] La nostra definizione semantica consta di due parti, *semantica statica* e *semantica dinamica*: la parte statica che si occupa della coerenza dei tipi verrà discussa in seguito.

```
{
    int x;
    x = 5;
      {                   (1)
       int y;
       y = x + 5;         (2)
       x = x + y;         (3)
      }                   (4)
    x = 2*x;              (5)
}
```

Per comodità, abbiamo numerato alcune linee, per poterle riferire nel seguito. Supponiamo che tale blocco venga eseguito a partire dallo stato vuoto, Ω. Lo stato al punto (1), al momento cioè dell'ingresso nel blocco annidato, è il seguente:

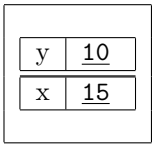


Dopo l'esecuzione dell'assegnamento (2), lo stato si presenta come segue:

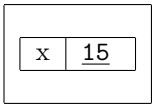


Si noti come l'ingresso nel blocco annidato abbia causato la creazione di un nuovo frame, destinato a contenere le variabili dichiarate, appunto, in tale blocco (nel caso specifico `y`).

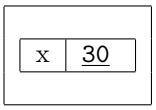
Dopo l'esecuzione dell'assegnamento (3) abbiamo il seguente stato:



All'uscita del blocco annidato, punto (4), il frame precedentemente creato cessa di esistere



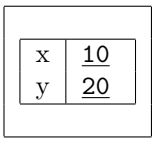
ed è in quest'ultimo stato che viene eseguito l'assegnamento (5), a seguito del quale abbiamo lo stato conclusivo



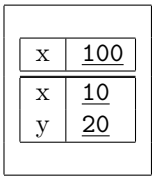
Vediamo infine un esempio che mostra come la semantica data consenta anche di trattare il caso di blocchi annidati in cui uno stesso identificatore viene ridefinito (purché non nello stesso blocco). Questa possibilità, pur se nella tradizione dei linguaggi imperativi quali Algol, Pascal e C, non viene contemplata in Java, dove la dichiarazione di una variabile `x` in un blocco annidato dentro un blocco più esterno in cui `x` è già stata dichiarata viene considerato come errore statico.

```
{
    int x = 10;
    int y = 20;
      {                   (1)
       int x = 100;       (2)
       y = y + x;         (3)
      }
    y = y + x;            (4)
}
```

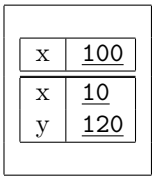
Supponiamo che tale blocco venga eseguito a partire dallo stato vuoto, Ω. Lo stato al punto (1), al momento cioè dell'ingresso nel blocco annidato, è il seguente:



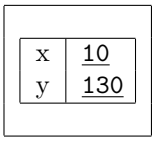
Dopo l'esecuzione della dichiarazione (2), lo stato si presenta come segue:



Si noti che la nuova variabile `x` fa parte del frame più recente, corrispondente al blocco annidato. In questo stato viene eseguito l'assegnamento (3), il cui effetto è rappresentato dal nuovo stato:



Si noti che, nell'espressione `y+x`, il riferimento ad `x` è un riferimento alla dichiarazione più recente per `x`. Dopo l'uscita dal blocco annidato, il frame corrispondente cessa di esistere e dunque il riferimento ad `x` in (4) è ora un riferimento alla dichiarazione di `x` del blocco più esterno. L'effetto dell'assegnamento (4) è dunque rappresentato dal seguente stato:



Esercizi

1. Valutare le seguenti liste di statements mediante il sottosistema $\rightarrow_{slist}$ a partire dallo stato $\sigma = \omega.\Omega$:

   (a) `int a; a=5; int b; b=3; int x=10;`

   (b) `int x; int y; int z;`

2. Valutare le seguenti liste di statements mediante il sottosistema $\rightarrow_{slist}$ a partire dallo stato $\sigma = \omega.\Omega$, mostrando graficamente lo stato dopo l'esecuzione di ciascun elemento della lista:

   (a) `int x=10; int y=20; {int y=30; x = x+y; y=2*y; y=2*y; x=x+y;}`

   (b) `int x=10; int y=20; {int x=40; int y=30; x = x+y; y=2*y;} y=2*y; x=x+y;`

   (c) `int x=10; int y=20; {int x=40; x = x+y; y=x; int y=30; x = x+y;} y=2*y; x=x+y;`

3. Valutare i seguenti blocchi a partire dallo stato $\sigma = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{10}\}.\Omega$:

   (a) `{int x=50; int z=50; y=x+2*z; x=x+y;}`

   (b) `{int z=50; int y=50; {int x=100; x = x+y; z=x;} x=x+z; y=x+1;}`

4. In riferimento ad una nozione di equivalenza per liste di statements analoga a quella introdotta nella definizione 5.4, dimostrare l'equivalenza delle liste $\mathtt{B}_1$ e $\mathtt{B}_2$ nei due casi seguenti:

(a) B_1 : `int a=5; int x;`
B_2 : `int x; int a=5;`

(b) B_1 : `int a; int b=10; a=5;`
B_2 : `int  a=5; int b=2*a;`

5.5 Programmi con classi e metodi di classe

Una metodologia molto utile per scrivere programmi complessi è la metodologia di programmazione per raffinamenti successivi. Tale metodologia può così essere descritta: Ogni volta che occorra esprimere un'operazione complessa, espressa cioè da una complessa struttura di controllo, si supponga di avere a disposizione nuovi operatori (con i quali formare espressioni) e nuovi comandi (con i quali esprimere nuove strutture di controllo), mediante i quali sia possibile realizzare tale operazione con una struttura di controllo più semplice di quella direttamente esprimibile nel linguaggio. La stesura del programma può così procedere speditamente esprimendo via via tutte le operazioni con strutture che fanno riferimento ad applicazioni di questi nuovi operatori (espressioni) e a nuovi comandi, introdotti dall'utente.

Una volta terminata la stesura del programma, si provvede a definire i nuovi comandi ed i nuovi operatori introdotti, con strutture che, a seconda della complessità, fanno riferimento o, ad ancora nuovi, ma più semplici comandi ed operatori, o, a comandi ed operatori del linguaggio. Se il linguaggio di programmazione mette a disposizione dei meccanismi di astrazione sul controllo che permettono di estendere il linguaggio con nuovi comandi o nuovi operatori, definiti dall'utente, allora il programma può essere scritto in accordo a tale metodologia. In questo modo il programma risulterà più semplice da scrivere, più compatto e comprensibile, più semplice da verificare e correggere. Un altro aspetto rilevante nella stesura di programmi è il seguente. Molto spesso programmi, con scopi diversi, hanno parti in comune o simili, cioè parti che calcolano la stessa funzione (ovvero che calcolano trasformazioni di stato equivalenti). Si osservino ad esempio i programmi di ordinamento degli array che utilizzano il metodo `swap` per scambiare i valori di due elementi di una variabile. I processi di definizione di operazioni complesse in termini di operazioni più semplici (raffinamento), e di generalizzazione di sequenze mediante parametri, sono aspetti diversi di un unico processo di astrazione.

In Java i meccanismi di astrazione sul controllo, che permettono di utilizzare la metodologia di programmazione per raffinamenti successivi, si chiamano *metodi* e possono essere di due tipi: di classe (o statici), sono i meccanismi di astrazione sul controllo dei linguaggi procedurali (detti in tali linguaggi *procedure* o *funzioni*), e istanza, sono i meccanismi di astrazione sul controllo tipici dei linguaggi object-oriented.

5.5.1 Metodi di classe

Pe definire sintassi e semantica dei metodi è necessario distinguere tra la *dichiarazione* di un metodo e la sua *invocazione*: la dichiarazione avviene una volta per tutte contestualmente alla dichiarazione della classe e definisce la funzionalità del metodo; l'invocazione o chiamata corrisponde all'utilizzo effettivo del metodo, alla esecuzione vera e propria delle azioni ad esso associate. In generale, invocazioni diverse dello stesso metodo possono avere effetti diversi poichè il metodo è parametrico, cioè definito in funzione di uno o più parametri che possono essere istanziati con valori diversi in chiamate diverse. Per semplicità, nel nostro linguaggio semplificato consideriamo metodi con un solo parametro.

In questo paragrafo introduciamo i metodi statici e le classi, mentre oggetti e metodi d'istanza verranno trattati nei prossimi paragrafi. Le classi contengono le definizioni dei metodi statici e d'istanza ,oltre alla definizione delle variabili d'istanza che vedremo più avanti. Di conseguenza, per definire la semantica dei metodi statici, abbiamo bisogno di definire la struttura che nello stato rappresenta le classi e di conseguenza la loro semantica. Tale struttura è una tabella del tipo:

`classe_1`	ρ_s^1	φ_1	ρ_m^1
`classe_2`	ρ_s^2	φ_2	ρ_m^2
....	....		
`classe_n`	ρ_s^n	φ_n	ρ_m^n

che associa al nome di ciascuna classe tre funzioni ρ_s per rappresentare i metodi statici , φ per rappresentare le variabili d'istanza e ρ_m per rappresentare i metodi d'istanza. Queste ultime due verranno introdotte nei paragrafi 5.6 e 5.6.3. Tale tabella può essere modellata mediante una funzione del tipo:

$$\texttt{Ide} \rightarrow \texttt{MethodEnv} \times \Phi \times \texttt{MethodEnv}.$$

Indicheremo con `ClassEnv` lo spazio delle funzioni del tipo appena definito ed utilizzeremo $\rho_c, \rho'_c, \ldots$ per indicare elementi di `ClassEnv`. Tutte le strutture introdotte sono funzioni rappresentate con frames ($\rho_c \in$ `ClassEnv`$, \rho_s, \rho_m \in$ `MethodEnv` e $\varphi \in \Phi$) pertanto le operazioni di selezione e modifica definite sui frames sono utilizzabili per manipolare tali strutture. Ad esempio se ρ_c è la tabella sopra $\rho_c($`classe_1`$)=< \rho_s^1, \varphi_1, \rho_m^2 >$. Le definizioni dei metodi siano essi statici che dinamici sono rappresentati da una tabella che associa ad ogni metodo `m` tutti le informazioni necessarie al momento della chiamata del metodo stesso, ovvero:

- il nome utilizzato per il parametro del metodo;
- il blocco che rappresenta le azioni intraprese dal metodo.

Le funzioni che rappresentano i metodi sono elementi di `MethodEnv` così definite:

$$\texttt{Ide} \rightarrow \texttt{Ide} \ \times \ \text{Block}.$$

Dunque, ρ_s è una funzione che, dato il nome di un metodo, restituisce una coppia in cui il primo elemento è il nome del parametro del metodo e il secondo elemento è il blocco, corpo, del metodo.

In questo paragrafo il linguaggio didattico viene esteso per permettere la definizione dei metodi statici e quindi delle classi. Il linguaggio così ottenuto può essere definito come il nucleo procedurale di Java. Sintatticamente, un programma nel linguaggio didattico è costituito da una sequenza di classi, che contengono definizioni di metodi statici, seguita da un blocco, cioè una lista di statement che è il (*corpo* del) programma principale [14]. Le espressioni ed i comandi del linguaggio sono estesi per invocare i metodi definiti. La sintassi del frammento di Java viene estesa come segue:

Prog::= **prog** { ClassDeclList {StatList} }
ClassDeclList ::= ClassDecl
| ClassDecl ClassDeclList

ClassDecl ::= **class** Ide StaticMetDefList

StaticMetDefList::= StaticMetDef StaticMetDefList |ε

StaticMetDef::= **public static** TypeR Ide$_m$ (Type Ide$_a$) Block

TypeR::= Type | void

Exp ::= ...| Ide$_c$.Ide$_m$(Exp) | ...

Com ::= ... | Ide$_c$.Ide$_m$(Exp) | ...

In una dichiarazione di classe si dichiara il nome della classe e per ora solo i metodi statici contenuti nella classe. Per ogni metodo si dichiara nell'ordine:

- il tipo del risultato oppure void. In questo caso il metodo definisce un nuovo comando altrimenti definisce un nuovo operatore.
- il nome del metodo Ide$_m$. Utilizzato per l'invocazione.
- il tipo e il nome del parametro (*formale*) Ide$_a$. Per semplicità diamo la semantica dei metodi con un unico parametro, anzicchè una lista di parametri.
- il corpo del metodo. Il blocco contenente la lista di statements che viene eseguita ogni volta che s'invoca il metodo.

[14] In Java il programma principale è definito in un metodo statico denominato *main* che deve necessariamente essere contenuto in una delle classi definite

Per l'invocazione del metodo si specifica:

- Il nome della classe in cui il metodo statico è definito (Ide_c).
- Il nome del metodo da invocare (Ide_m)
- l'espressione valore del parametro (*attuale*)

Consideriamo il seguente esempio di programma nel linguaggio didattico:

```
prog { class Num {
            public static int fact (int p) {
                 if (p==0) ...}
            public static boolean pari (int x) {...}
    }
      class Geometria{
           public static void faiCerchio(int r) {...}}

    { int x=4; int y=Num.fact(x);}
 }
```

In questo esempio vengono definite due classi `Num` con due metodi `fact` e `pari` e `Geometria` con un unico metodo `faiCerchio`. Un esempio di invocazione di metodo è contenuto nel programma, `Num.fact(x)` che compare come espressione nella dichiarazione della variabile `y`. in questa invocazione abbiamo la variabile `x`, parametro attuale, mentre il parametro formale che compare nella dichiarazione del metodo è ha nome `p`. L'ambiente delle classi ρ_c di tale programma può essere rappresentato dalla seguente tabella:

`Num`	ρ_s^{Num}	ω	ω
`Geometria`	$\rho^{G}eometria_s$	ω	ω

A questo punto dobbiamo definire la semantica del programma, della dichiarazione delle classi che costruisce l'ambiente delle classi (ρ_c) e dell'invocazione dei metodi statici.

La semantica di un programma è la semantica della lista di statment che costituiscono il corpo del programma a partire dallo stato costituito dall'ambiente delle classi e dallo stack che contiene un solo frame vuoto ($\omega.\Omega$).

$$\frac{\langle \texttt{Cdl},\ \omega\rangle \rightarrow_{classdecl} \rho_c \qquad \langle \texttt{Stat\_List}, \langle \rho_c, \omega.\Omega\rangle\rangle \rightarrow_{slist} \langle\sigma\rangle}{\langle \texttt{prog\{Cdl \{Stat\_List\}\}}\rangle \rightarrow_{prog} \langle\sigma\rangle} \qquad (prog)$$

Vediamo la definizione del sottosistema $\rightarrow_{classdecl}$ utilizzato in (*prog*). La prima regola qui definita è semplificata per il caso in cui i soli metodi statici sono definiti. Nei paragrafi successivi tali regole saranno ridefinite per considerare il caso generale.

$$\frac{\langle \texttt{SMdl}, \omega\rangle \rightarrow_{methoddecl} \rho_s}{\langle \texttt{class c \{ SMdl}\}, \rho_c\rangle \rightarrow_{classdecl} \rho_c[^{(\rho_s,\omega,\omega)}/_{\texttt{c}}]} \qquad (classdecl)$$

$$\frac{\langle \texttt{Cd},\ \rho_c\rangle \rightarrow_{classdecl} \rho_c' \qquad \langle \texttt{Cdl}, \rho_c'\rangle \rightarrow_{classdecl} \rho_c''}{\langle \texttt{Cd}\quad \texttt{Cdl}, \rho_c\rangle \rightarrow_{classdecl} \rho_c''} \qquad (classdecl_{list})$$

Lo scopo del sottosistema $\rightarrow_{methoddecl}$ è di creare la struttura ρ_s che contiene una rappresentazione dei metodi della classe.

$$\langle \texttt{public void m(T x) B}, \rho_s \rangle \rightarrow_{methoddecl} \rho_s[^{(\texttt{x,B})}/_{\texttt{m}}] \qquad (methoddecl)$$

$$\frac{\langle \texttt{Md}, \ \rho_s \rangle \rightarrow_{methoddecl} \rho'_s \qquad \langle \texttt{Mdl}, \rho'_s \rangle \rightarrow_{methoddecl} \rho''_s}{\langle \texttt{Md} \quad \texttt{Mdl}, \rho_s \rangle \rightarrow_{methoddecl} \rho''_s} \qquad (methoddecl_{list})$$

5.5.2 Invocazione dei metodi di classe

La semantica dell'invocazione di metodi, che estende il sistema $\rightarrow_{com}$ può ora essere definita. Consideriamo per ora il caso di un metodo statico che calcola un valore (TypeR diverso da void), dal momento che a causa della semplicità del linguaggio a questo punto, lo stato calcolato, altrimenti, è semplicemente lo stato di partenza. Le semplificazioni che causano questa situazione sono due, il non considerare nello stato l'input/output, e il non considerare gli oggetti. La semantica dei metodi statici (che non calcolano un valore) verrà ripresa dopo aver introdotto gli oggetti.

Per l'invocazione dei metodi che calcolano un valore, bisogna prima di tutto affrontare il problema del valore calcolato e stabilire come questo valore debba essere memorizzato nello stato. La soluzione adottata, valore calcolato viene memorizzato in una variabile speciale, di nome `retval`, definita nello stato di valutazione del corpo del metodo, alla quale viene assegnato un valore quando si valuta un comando `return Exp`. La prima regola definita mostra che la valutazione dell'invocazione del metodo calcola il valore di `retval` nello stato risultante dalla valutazione del corpo del metodo invocato. Il corpo del metodo viene valutato in uno stato σ' costituito da un solo frame che contiene un legame per il parametro, formale legato al valore dell'attuale. Il formale è l'identificatore `x` reperito in ρ_s e l'attuale è l'espressione `E` a cui il metodo è applicato. La semantica è formalmente definita dalle seguenti regole:

$$\frac{\begin{array}{c}\rho_c(\texttt{c}) = (\rho_s, \varphi, \rho_m) \qquad \rho_s(\texttt{m}) = (\texttt{x}, \texttt{B}) \qquad \langle \texttt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \ \texttt{v} \\ \sigma' = \omega[^{\texttt{v}}/_{\texttt{x}}, ^{\varpi}/_{\texttt{retval}}].\Omega \qquad \langle \texttt{B}, \langle \rho_c, \sigma', \rangle \rangle \rightarrow_{com} \ \sigma''\end{array}}{\langle \texttt{c.m(E)}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{com} \sigma''(\texttt{retval})} \qquad (exp_{StatCall})$$

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \ \texttt{v}}{\langle \texttt{return E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{com} \sigma[^{\texttt{v}}/_{\texttt{retval}}]} \qquad (exp_{return})$$

Le regole per le espressioni devono essere modificate per tenere conto del nuovo stato che ora ha due componenti. Le riportiamo di seguito senza ulteriori commenti dal momento che sono ottenute semplicemente sostituendo il nuovo stato $\langle \rho_c, \sigma \rangle$ al posto di φ nelle regole definite nel paragrafo 3.2.

$$\langle \mathtt{n}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad (exp_n)$$

$$\frac{\sigma(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{ide})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\langle \mathtt{E}+\mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_+)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\langle \mathtt{E}*\mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_*)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\langle \mathtt{E}\text{-}\mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_-)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \; div \; \underline{\mathtt{n}}'}{\langle \mathtt{E} \; / \; \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{div})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \; mod \; \underline{\mathtt{n}}'}{\langle \mathtt{E} \; \% \; \mathtt{E}', \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{mod})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}}{\langle (\mathtt{E}), \langle \rho_c, \sigma \rangle \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{()})$$

Nella regola semantica data sopra risulta che lo stato di valutazione del corpo del metodo è lo stato $\langle \rho_c, \omega[^{\mathtt{v}}/_{\mathtt{x}}, ^{\varpi}/_{\mathtt{retval}}].\Omega \rangle$. Di conseguenza tutti gli identificatori contenuti in ρ_c sono accessibili ed inoltre è accessibile il parametro formale `x` e sarebbe accessibile anche la variabile speciale `retval` anche se non abbiamo mai bisogno di usarla perchè l'assegnamento di un valore a tale variabile viene fatto con il comando `return`. Non risultano invece accessibili le variabili definite nello stato dell'invocazione cioè (σ). Questo è conseguenza della politica di scoping di Java che è *statica*. Linguaggi in cui lo stack di valutazione del corpo del programma viene costruito estendendo lo stato σ di invocazione sono linguaggi a scoping dinamico. La maggior parte dei linguaggi di programmazione adotta uno scoping statico che consente di definire programmi più comprensibili e possibilmente corretti.

5.6 Programmazione con gli oggetti

I linguaggi Object-oriented modellano la porzione di mondo reale che deve essere automatizzata privilegiando le entità (oggetti) mentre gli altri linguaggi privilegiano le operazioni (funzioni) cioè la computazione che deve essere svolta dal calcolatore. Gli oggetti dei linguaggi O-O hanno delle proprietà e delle operazioni (metodi d'istanza). Le proprietà hanno un nome , un valore e un tipo. Le operazioni, come i metodi statici, hanno un nome un tipo del risultato, dei parametri e un corpo. Tutti gli oggetti dello stesso tipo fanno parte della medesima *classe*, che può dunque essere vista come una categoria di oggetti. Nel linguaggio che prendiamo in considerazione (un frammento di Java) la definizione di una classe prevede:

- le dichiarazioni delle cosiddette *variabili istanza*, che permettono di definire le proprietà dell'oggetto, associando ad ogni oggetto della classe un proprio *stato*;
- le definizioni dei *metodi d'istanza*, che definiscono le operazioni che possono essere eseguite su ogni oggetto della classe.
- le definizioni dei metodi statici che abbiamo visto nel paragrafo 5.5.1.

Una volta definita una classe (si veda il paragrafo 5.5), si possono dichiarare variabili i cui valori sono oggetti di quella classe. In realtà, in Java tutti i tipi di dato non elementari, a partire dal tipo delle stringhe (sequenze di caratteri), sono trattati con il meccanismo delle classi. Così, ad esempio, se `Foo` è il nome di una classe, una dichiarazione del tipo

```
Foo obj1;
```

definisce una nuova variabile il cui scopo è quello di riferire un nuovo oggetto della classe `Foo`. Una dichiarazione come la precedente, ha lo stesso effetto di una dichiarazione di variabile come quelle viste nella sezione 5.4, ovvero di introdurre nello stato corrente una nuova associazione per il nome `myobj`. Non trattandosi di un tipo elementare, dobbiamo ancora chiarire quale sia il valore associato a tale nome. Osserviamo che una dichiarazione come questa, tuttavia, non comporta la creazione vera e propria dell'oggetto associato alla variabile `obj1`: tale creazione avviene invece tramite l'uso della primitiva `new`. Intuitivamente, l'esecuzione di

```
new Foo
```

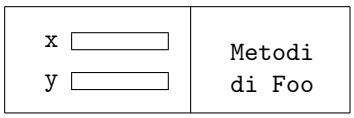
ha l'effetto di creare la vera e propria rappresentazione di un nuovo oggetto della classe `Foo`, con il suo proprio stato (ovvero con le sue proprie variabili istanza) ed i suoi metodi. Molto spesso l'inizializzazione di un oggetto avviene contestualmente alla dichiarazione di una variabile oggetto, come in

```
Foo obj1 = new Foo;          (*)
```

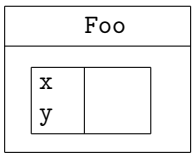
Nella sezione 5.6.1 daremo la semantica formale di tutti questi concetti: limitiamoci per ora ad una rappresentazione grafica di ciò che accade al momento di una dichiarazione ed inizializzazione come (∗). Per semplicità, consideriamo per il momento che la classe `Foo` sia già stata definita e che essa preveda la presenza di due variabili istanza, `x` e `y`, entrambe di tipo `int`. La definizione di `Foo` conterrà in generale anche dei metodi, che per il momento prendiamo in considerazione solo marginalmente. La dichiarazione (∗) ha allora i seguenti effetti.

1. Si crea la rappresentazione del nuovo oggetto, ovvero una struttura che:
   - contenga lo stato proprio dell'oggetto, costituito dalle sue variabili istanza `x` e `y`;
   - consenta di accedere quando necessario ai metodi definiti nella classe di appartenenza del nuovo oggetto.

   Una tale struttura può essere del tipo rappresentato di seguito:

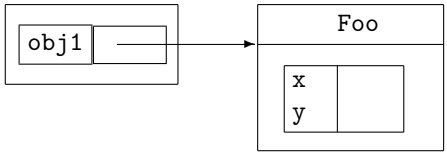


Si noti che lo stato non è altro che un frame, secondo la terminologia introdotta nella sezione 4.2. Un'altra possibile rappresentazione dell'oggetto è quella descritta nella seguente figura:



Anche in questo caso abbiamo un frame che rappresenta lo stato dell'oggetto; non abbiamo invece una copia dei metodi della classe di appartenenza, quanto piuttosto il nome della classe stessa. Ciò presuppone che le definizioni delle classi diano luogo ad una rappresentazione delle stesse che preveda la possibilità di reperire un metodo della classe a partire dal nome della classe stessa. Questa scelta non può chiaramente essere fatta anche per le variabili istanza, in quanto esse devono essere distinte per ogni oggetto della classe. I metodi, essendo entità che una volta definiti non possono essere modificati, possono invece essere mantenuti in una struttura comune a tutti gli oggetti della medesima classe. La scelta che adottiamo in queste note è la seconda, più vicina ad una reale implementazione del linguaggio Java. Nella prossima sezione vedremo in che modo le definizioni delle classi danno luogo alla struttura cui abbiamo appena accennato.

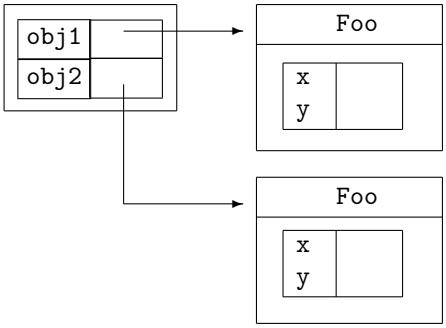
2. Nello stato corrente, si associa la rappresentazione appena costruita alla variabile `obj1`: ciò avviene associando a quest'ultima un *riferimento* all'oggetto appena creato, rappresentato in figura mediante una freccia (in seguito, definiremo formalmente che cosa sono i riferimenti).



Come ulteriore esempio, immaginiamo, a partire dalla situazione precedente, una ulteriore dichiarazione di variabile

`Foo obj2 = new Foo;`

La situazione conseguente alla esecuzione di tale dichiarazione sarà quella descritta nella figura:



in cui lo stato contiene anche una associazione per `obj2`, la cui rappresentazione è del tutto analoga a quella di `obj1`, seppure da essa distinta.

Le rappresentazioni degli oggetti, ovvero le strutture come quelle associate alle variabili `obj1` e `obj2` dell'esempio precedente, vengono mantenute in una opportuna struttura semantica, denominata *heap*: dunque ogni invocazione della primitiva `new` richiede la creazione di una nuova struttura e la sua memorizzazione nello *heap*.

Da un punto di vista formale, dobbiamo definire con precisione:

- le rappresentazioni degli oggetti: queste possono essere modellate mediante coppie del tipo

  $$(\mathtt{C}, \varphi)$$

  dove φ è un frame e `C` è il nome della classe dell'oggetto. Ad esempio, la rappresentazione inizialmente associata alla variabile `obj1` dell'esempio precedente è una coppia

  $$(\mathtt{Foo}, \{\mathtt{x} \mapsto \varpi, \mathtt{y} \mapsto \varpi\})$$

  (assumendo che i valori delle variabili istanza siano indefiniti).

- lo *heap*, che può essere modellato mediante una tabella (una funzione a dominio finito) che associa rappresentazioni di oggetti (ovvero coppie del tipo $(\mathtt{c}, \varphi)$ a riferimenti ad oggetti (rappresentati da frecce nelle figure precedenti). I riferimenti sono presi da un insieme opportuno, che chiameremo `Loc`. Dunque uno *heap* è una funzione del tipo

  $$\mathtt{Loc} \rightarrow \mathtt{Ide} \times \Phi$$

  In seguito, denoteremo con `Heap` l'insieme dei possibili heap ed utilizzeremo $\zeta, \zeta', \ldots$ per denotare elementi di `Heap`.

Come abbiamo già detto nei paragrafi precedenti la definizione di una classe permette di definire il tipo di una categoria di oggetti con le sue proprietà, variabili d'istanza, e le sue operazioni, metodi d'istanza nonchè i metodi statici . Consideriamo un esempio di definizione di classe.

```
class Foo
{
  int x;
  int y;
  <metodi della classe>
}
```

Dal punto di vista semantico, l'effetto di una dichiarazione come la precedente è di mantenere in una opportuna struttura semantica le informazioni che saranno necessarie al momento della creazione di un nuovo oggetto della classe appena definita. Tali informazioni, associate al nome della classe, sono un frame che rappresenta le variabili istanza e opportune rappresentazioni ρ_s e ρ_m dei metodi. Si ricorda che struttura semantica che rappresenta la classi è una tabella del tipo:

`classe_1`	ρ_s^1	φ_1	ρ_m^1
`classe_2`	ρ_s^2	φ_2	ρ_m^2
....	....		
`classe_n`	ρ_s^n	φ_n	ρ_m^n

dove φ_i è il frame che rappresenta le variabili d'istanza di $\mathtt{classe_i}$ in cui i valori associati agli identificatori sono, per tutti i tipi, il valore indefinito (ϖ) e ρ_s^i e $\rho_m^i \in \mathtt{MethodEnv}$ sono l'ambiente dei metodi statici e d'istanza della classe i. Si noti che, al momento della creazione di un nuovo oggetto mediante una dichiarazione del tipo

$$\mathtt{Foo\ myobj} = \mathtt{new\ Foo};$$

la coppia $(\mathtt{Foo}, \varphi)$ che viene aggiunta allo heap come rappresentazione di `myobj`, viene costruita proprio a partire da quanto associato all'identificatore `Foo` in ρ_c: in particolare, il frame φ non è altro che la seconda componente della tripla $\rho_c(\mathtt{Foo})$.

5.6.1 Semantica dei programmi nel linguaggio semplificato

La sintassi del linguaggio può ora essere estesa con le regole complete per la dichiarazione delle classi. Per semplicità, assumiamo che tutte le variabili istanza delle classi siano dichiarate come `public`. In realtà nella buona pratica di programmazione è bene che le variabili istanza degli oggetti siano dichiarate come `private` e siano accessibili solo attraverso i metodi della classe. Si noti comunque che il controllo sull'uso privato delle variabili può essere demandato ad un controllo statico senza coinvolgere la semantica dinamica che stiamo trattando.

La sintassi del linguaggio didattico viene estesa come segue:

Prog::= **prog** { ClassDeclList {StatList} }
ClassDeclList ::= ClassDecl
| ClassDecl ClassDeclList

ClassDecl ::= **class** Ide {StaticMetDefList IVarDecList MetDeflist}

StaticMetDefList::= ...
MetDefList::= ...
IVarDecList::= IVarDecl
| IVarDecl IVarDecList
IVarDecl::= **public** Type Ide

TypeR::= Type | void

Exp ::= ...| Ide.Ide | ...

Com ::= ... | Ide.Ide=Exp | ide = new Ide | ...

Dec ::= Ide Ide; | Ide Ide= new Ide;

Le nuove produzioni per la categoria sintattica `Com` consentono di esprimere, rispettivamente:

- la modifica di una variabile istanza di un oggetto: in `x.y = E;`, `x` deve fare riferimento ad un oggetto e `y` deve essere l'identificatore di una variabile istanza dichiarata nella classe di appartenenza di tale oggetto (ad esempio `myobj.x = 5;`);
- l'assegnamento di un nuovo oggetto ad una variabile oggetto (ad esempio `myobj = new Foo;`)

La nuova produzione per la categoria sintattica `Exp` consente di rappresentare espressioni che dipendono dal valore di variabili istanza di oggetti. Infine, le nuove produzioni per la categoria sintattica `Dec` consentono di dichiarare variabili oggetto, eventualmente inizializzandole mediante l'operazione `new`.

Esempio 5.5 Consideriamo un semplice programma:

```
prog {
    class Foo
        {
            public int x;
            public int y;
        }
    class Fie
        {
            public int x;
        }
```

```
{
    Foo o1 = new Foo;
    Fie o2 = new Fie;
    o1.x = 5;
    o2.x = 10;
    o1.y = o1.x + o2.x;
}
}
```

Il lettore attento avrà notato che il programma non è sintatticamente corretto rispetto alla grammatica data, poiché le classi `Foo` e `Fie` hanno una sequenza vuota di dichiarazioni di metodi. Lasciamo al lettore come esercizio la modifica della grammatica che consenta di definire classi senza variabili istanza o senza metodi. ■

La regola semantica di un programma deve essere ridefinita per includere lo heap (ζ) terza componenente dello stato , inizialmente vuota. Formalmente:

$$\frac{\langle \texttt{Cdl},\ \omega\rangle \rightarrow_{classdecl} \rho_c \qquad \langle \texttt{Stat\_List}, \langle \rho_c, \omega.\Omega, \omega\rangle\rangle \rightarrow_{slist} \langle \sigma, \zeta\rangle}{\langle \texttt{prog\{Cdl \{Stat\_List\}\}}\rangle \rightarrow_{prog} \langle \sigma, \zeta\rangle} \qquad (prog)$$

Si noti che le configurazioni del sottosistema $\rightarrow_{com}$ sono cambiate rispetto a quelle riportate nella sezione 5.2. Adesso un blocco va eseguito non più rispetto a un semplice stato, ma rispetto ad una tripla $\langle \rho_c, \sigma, \zeta\rangle$ dove:

- ρ_c è l'ambiente delle classi dichiarate all'inizio del programma;
- σ è una pila di frame (inizialmente vuota) necessaria per modellare l'annidamento dei blocchi, come visto nella sezione 4.2;
- ζ è uno heap (inizialmente vuoto) che associa locazioni ad oggetti, necessario per il trattamento degli oggetti come visto nella sezione **??**.

Da ora in poi chiameremo *stato* una tripla siffatta. Inoltre, le configurazioni terminali del sistema $\rightarrow_{com}$ sono adesso una coppia del tipo $\langle \sigma, \zeta\rangle$, che rappresenta le componenti modificabili dello stato (si noti infatti che l'ambiente delle classi, una volta costruito, non viene più modificato).

Le regole per il sistema $\rightarrow_{com}$ viste nella Sezione 5.2 vanno opportunamente riviste a seguito della modifica dello stato. Rivediamo come esempi la regola per l'assegnamento e la regola (com_{if-tt}), lasciando per esercizio la modifica delle altre.

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{v}}{\langle \texttt{x=E;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma[^{\texttt{v}}/_{\texttt{x}}], \zeta\rangle} \qquad (com_{=})$$

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} tt \qquad \langle \texttt{C1}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle}{\langle \texttt{if (E) C1 else C2}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle} \qquad (com_{if-tt})$$

Come potevamo aspettarci, anche le configurazioni del sistema $\rightarrow_{exp}$ sono modificate, rispetto a quelle viste nella sezione 4.1.1, per tener conto del nuovo stato. Anche per esse, vediamo un esempio di regola modificata:

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{n} \qquad \langle \mathtt{E}', \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{n}' \qquad \mathtt{m} = \mathtt{n} + \mathtt{n}'}{\langle \mathtt{E}{+}\mathtt{E}', \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{m}} \qquad (exp_{+})$$

Più significativa è la modifica dell'unica regola di $\rightarrow_{exp}$ in cui la componente stato gioca un ruolo cruciale:

$$\frac{\sigma(\mathtt{x}) = \mathtt{n}}{\langle \mathtt{x}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{n}} \qquad (exp_{ide})$$

Anche le regole del sottosistema $\rightarrow_{slist}$ vanno opportunamente riviste. Consideriamo ad esempio la regola $(slist_{list})$ che diventa:

$$\frac{\langle \mathtt{S}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma'', \zeta'' \rangle \qquad \langle \mathtt{Slist}, \langle \rho_c, \sigma'', \zeta'' \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle}{\langle \mathtt{S\ \ Slist}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle} \qquad (slist_{list})$$

Per il sottosistema $\rightarrow_{classdecl}$ utilizzato in $(prog)$ deve essere ridefinita solo la regola classdecl.

$$\frac{\langle \mathtt{SMdl}, \omega \rangle \rightarrow_{methoddecl} \rho_s \qquad \langle \mathtt{IVdl}, \omega \rangle \rightarrow_{instdecl} \varphi \qquad \langle \mathtt{Mdl}, \omega \rangle \rightarrow_{methoddecl} \rho_m}{\langle \mathtt{class\ c\ \{SMdl\ \ IVdl\ \ Mdl\}}, \rho_c \rangle \rightarrow_{classdecl} \rho_c[^{(\varphi, \rho_m)}/_{\mathtt{c}}]} \qquad (classdecl)$$

Nella regola $(classdecl)$, a partire dalla sequenza di dichiarazioni di variabili istanza `IVdl` viene costruito un frame φ (premessa $\langle \mathtt{IVdl}, \omega \rangle \rightarrow_{instdecl} \varphi$) che verrà utilizzato per rappresentare lo stato interno di ciascun oggetto della classe. Inoltre, a partire dalla sequenza di dichiarazioni di metodi `Mdl` viene costruito l'ambiente dei metodi ρ_m che associa a ciascun metodo la sua rappresentazione (premessa $\langle \mathtt{Mdl}, \omega \rangle \rightarrow_{methoddecl} \rho_m$). Il sottosistema $\rightarrow_{methoddecl}$ verrà discusso nella sezione 5.6.3.
Il sottosistema $\rightarrow_{instdecl}$ è una semplificazione del sottosistema $\rightarrow_{dec}$, le cui regole sono le seguenti.

$$\langle \mathtt{public\ T\ x;}, \varphi \rangle \rightarrow_{instdecl} \varphi[^{\varpi}/_{\mathtt{x}}] \qquad (instdecl)$$

$$\frac{\langle \mathtt{IVd},\ \varphi \rangle \rightarrow_{instdecl} \varphi' \qquad \langle \mathtt{IVdl}, \varphi' \rangle \rightarrow_{instdecl} \varphi''}{\langle \mathtt{IVd\ \ IVdl}, \varphi \rangle \rightarrow_{instdecl} \varphi''} \qquad (instdecl_{list})$$

La semplificazione rispetto a $\rightarrow_{dec}$ è dovuta al fatto che l'unica componente dello stato richiesta è il frame costruito a partire dalle dichiarazioni delle variabili istanza.

In riferimento al programma dell'esempio 5.5, l'ambiente delle classi ρ_c dopo la valutazione delle dichiarazioni di `Foo` e `Fie` è la seguente funzione:

$$\rho_c = \{\mathtt{Foo} \mapsto (\varphi_{Foo}, \rho_m^{Foo}), \mathtt{Fie} \mapsto (\varphi_{Fie}, \rho_m^{Fie})\}$$

dove i frames φ_{Foo} e φ_{Fie} sono:

$$\varphi_{Foo} = \{\mathtt{x} \mapsto \varpi, \mathtt{y} \mapsto \varpi\}$$
$$\varphi_{Fie} = \{\mathtt{x} \mapsto \varpi\}$$

Le strutture semantiche $\rho_m^{Foo}, \rho_m^{Fie} \in$ `MethodEnv` sono solo indicate: come vedremo nella sezione 5.6.3 in questo semplice caso in cui le due classi non definiscono alcun metodo, tali strutture coincidono con ω.

Possiamo rappresentare graficamente la situazione come segue:

ρ_c

`Foo`	ω	`x` ϖ `y` ϖ	ω
`Fie`	ω	`x` ϖ	ω

Per procedere con la semantica del blocco principale (seconda premessa della regola (*prog*)), dobbiamo formalizzare il trattamento della dichiarazione di variabili oggetto e dell'operazione `new`, visti informalmente nelle sezioni precedenti.
Consideriamo innanzitutto l'operazione `new`: il suo effetto è quello di costruire la rappresentazione del nuovo oggetto, associarla ad un nuovo riferimento nello heap e restituire, oltre allo heap modificato, anche tale riferimento. Dunque il risultato della valutazione di `new c` è una coppia (l, ζ), dove l è il riferimento nel nuovo heap ζ alla rappresentazione creata. Il riferimento l deve essere distinto da tutti i riferimenti già in uso nello heap, e a questo scopo utilizziamo una funzione *newloc* che, dato uno heap, restituisce un riferimento non in uso. Formalmente, la regola per la valutazione di `new c` è la seguente:

$$\frac{\rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad l = newloc(\zeta)}{\langle \mathtt{new\ c}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{new} \langle l, \zeta[^{(\mathtt{c},\varphi)}/_l] \rangle} \qquad (new)$$

Quando si valuta `new c`, dove `c` è il nome della classe, occorre accedere alla descrizione di `c` nell'ambiente delle classi - premessa $\rho_c(\mathtt{c}) = (\varphi, \rho_m)$ - per poi aggiungere nello heap la rappresentazione del nuovo oggetto, costituito dalla coppia $(\mathtt{c}, \varphi)$.

Possiamo ora dare la semantica formale della dichiarazione di una variabile oggetto, estendendo il sistema di transizioni $\rightarrow_{dec}$. Nelle regole che seguono `c` sta per il nome di una classe.

$$\frac{\sigma = \varphi.\sigma'}{\langle \mathtt{c\ x;}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{dec} \langle \varphi[^{null}/_{\mathtt{x}}].\sigma', \zeta \rangle} \qquad (dec_{obj-1})$$

$$\frac{\sigma = \varphi.\sigma' \qquad \langle \mathtt{new\ c}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{new} \langle l, \zeta' \rangle}{\langle \mathtt{c\ x\ =\ new\ c;}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{dec} \langle \varphi[^{l}/_{\mathtt{x}}].\sigma', \zeta' \rangle} \qquad (dec_{obj-2})$$

Come si vede, in entrambi i casi lo stato viene modificato aggiungendo, nel frame più recente, una nuova coppia per la variabile oggetto appena dichiarata, cui viene associato un riferimento. Quest'ultimo è il riferimento speciale `null` nel caso in cui non venga creata una nuova rappresentazione attraverso l'operazione `new`.

Osserviamo che le configurazioni terminali del sottosistema $\rightarrow_{dec}$ sono ora una coppia in cui:

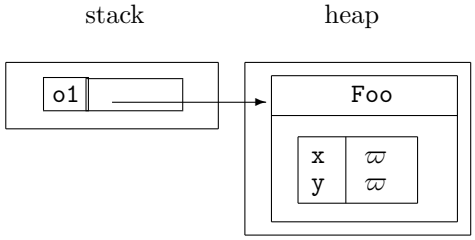
- la prima componente è la pila di frame dove al frame più recente è stata aggiunta una associazione per la variabile oggetto dichiarata (come accade nel caso delle variabili semplici);
- la seconda componente è lo heap, che può risultare modificato a seguito della inizializzazione della variabile oggetto appena dichiarata (nel caso della regola (dec_{obj-2})).

Queste considerazioni implicano ovviamente che anche tutte le regole già definite per $\rightarrow_{dec}$ vanno riviste opportunamente. Riportiamo come esempio la regola (dec_{var-1})

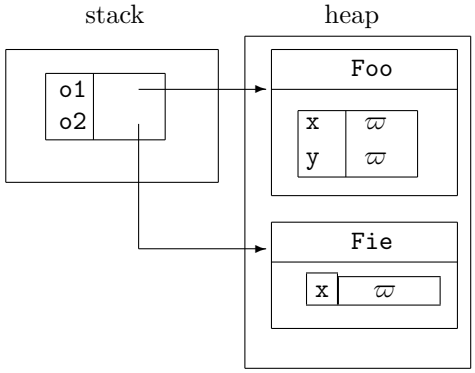
$$\frac{\sigma = \varphi.\sigma'' \qquad \sigma' = \varphi[^{\varpi}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x;}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{dec} \langle \sigma', \zeta \rangle} \qquad (dec_{var-1})$$

e lasciamo per esercizio la modifica delle altre regole.

In riferimento all'esempio 5.5, possiamo dare una rappresentazione grafica dello stato successivo alla dichiarazione `Foo o1 = new Foo;` come segue, dove gli elementi di `Loc` vengono rappresentati mediante delle frecce.



A seguito della seconda dichiarazione `Fie o2 = new Fie;`, lo stato precedente evolve come rappresentato nella seguente figura:



Durante l'esecuzione di un programma è possibile associare ad una variabile oggetto precedentemente dichiarata un nuovo oggetto (purché della stessa classe). Un assegnamento del tipo `o = new c;`, dove `o` è l'identificatore di un oggetto di classe `c`, ha proprio questo scopo. Per trattare tale situazione, estendiamo il sistema $\rightarrow_{com}$ con la nuova regola seguente.

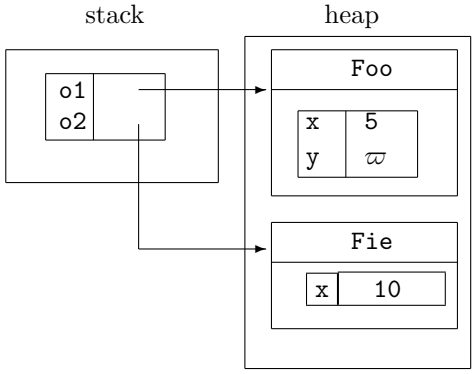
$$\frac{\langle \mathtt{new\ c}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{new} \langle l, \zeta' \rangle}{\langle \mathtt{o = new\ c;}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma[^{l}/_{\mathtt{o}}], \zeta' \rangle} \qquad (com_{=new})$$

Procediamo con la semantica dell'assegnamento di valori a variabili istanza (come in `o1.x=5` dell'esempio). Dobbiamo estendere opportunamente il sistema $\rightarrow_{com}$ a questo caso.

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{v} \qquad \zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi) \qquad \varphi' = \varphi[^{\mathtt{v}}/_{\mathtt{x}}]}{\langle \mathtt{o.x = E;}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma, \zeta[^{(\mathtt{c},\varphi')}/_{\sigma(\mathtt{o})}] \rangle} \qquad (com_{=2})$$

A partire dall'identificatore dell'oggetto in questione (l'identificatore `o` nella regola), si accede alla sua rappresentazione $\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi)$. Il frame φ associato all'oggetto viene quindi modificato associando alla variabile istanza `x` il valore `v` dell'espressione `E` calcolato nello stato corrente. Tale modifica si riflette nello heap della configurazione di arrivo con l'aggiornamento della seconda componente della coppia $\sigma(\mathtt{o})$ associata nello heap all'oggetto.

Consideriamo di nuovo l'esempio 5.5 e rappresentiamo lo stato successivo ai primi due assegnamenti `o1.x=5; o2.x=10;`.

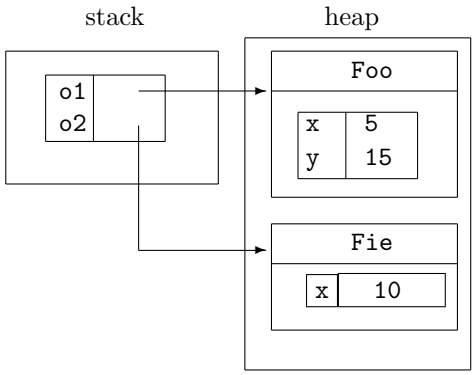


Le variabili istanza possono essere utilizzate anche nelle espressioni, come `o1.x` e `o2.x` nell'assegnamento `o1.y = o1.x + o2.x;` dell'esempio precedente: dobbiamo quindi estendere anche il sottosistema $\rightarrow_{exp}$.

$$\frac{\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi)}{\langle \mathtt{o.x}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \varphi(\mathtt{x})} \qquad (exp_{ref})$$

Il valore della variabile istanza `x` dell'oggetto `o` viene prelevato nel frame associato a $\sigma(\mathtt{o})$ nello heap corrente.

Concludendo l'esempio 5.5, lo stato dopo l'ultimo assegnamento `o1.y = o1.x + o2.x` è rappresentato dalla seguente figura:



5.6.2 Invocazione dei metodi di classe che non calcolano un valore

In questa sezione definiamo la semantica dell'invocazione di un metodo statico che non calcola nessun valore come risultato ma produce una modifica dello stato, nell'unica componenente che può essere modificata da un'invocazione di metodo, cioè lo heap.

$$\frac{\rho_c(\mathtt{c}) = (\rho_s, \varphi, \rho_m) \qquad \rho_s(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \qquad \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \omega[^{\mathtt{v}}/_{\mathtt{x}}].\Omega \qquad \langle \mathtt{B}, \langle \rho_c, \sigma', \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta' \rangle}{\langle \mathtt{c.m(E)}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma, \zeta' \rangle} \quad (com_{call})$$

In questo caso l'invocazione è un comando, dal momento che non calcola un valore ma solo una modifica dello stato, infatti lo stato restituito alla fine dell'invocazione ha come stack lo stack al momento dell'invocazione (σ) e come heap (ζ'), cioè lo heap risultante dall'esecuzione del corpo del metodo (B) nello stato che ha come seconda componente uno stack che contiene un unico frame che a sua volta contiene un unico legame per il parametro formale (x) con associato il valore del parametro attuale v.

5.6.3 Metodi istanza

Come abbiamo già accennato nel paragrafo **??** i metodi possono essere di classe (anche detti statici) o istanza. La differenza tra i metodi statici e quelli istanza sta nel fatto che i primi non hanno un oggetto di riferimento come i secondi. I metodi istanza hanno accesso alle variabili d'istanza dell'oggetto di riferimento e sono operazioni su di esso che possono produrre un valore e/o una modifica di stato. L'oggetto di riferimento è un parametro implicito, che non viene dichiarato che ha come nome (formale) `this`, come tipo la classe e come valore (attuale) l'oggetto stesso. Inoltre, le restrizioni sul nostro linguaggio impongono che gli unici effetti che i metodi possono avere siano la modifica dello stato degli oggetti coinvolti [15].

5.6.4 Dichiarazione dei metodi

La dichiarazione dei metodi istanza è identica a quella de metodi di classe. La richiamiamo brevemente:

```
MethodDeclList ::= MethodDecl |
                   MethodDecl MethodDeclList

MethodDecl ::= public TypeR Ide ( Type Ide ) Block
```

Anche in questo caso consideriamo metodi con un solo parametro esplicito. Di seguito un esempio di dichiarazione di classe in cui è presente un metodo.

Esempio 5.6 Consideriamo la seguente definizione della classe `Incr`. La specifica di this nel corpo del metodo può essere omessa in quanto inserita automaticamente dal compilatore.

```
class Incr
    {
        public int x;
        public void inc (int y)
           {
             this.x = this.x+y;
           }
    }
```

■

La semantica della dichiarazione di metodo istanza non differisce da quella della dichiarazione di metodo di classe. Con l'unica osservazione che abbiamo deciso di mantenere due ambienti separati ρ_s per i metodi di classe e ρ_m per i metodi d'istanza.

Supponendo che `o` sia un oggetto della classe `Incr` dell'esempio precedente, la chiamata del metodo `inc` di `o` deve avvenire istanziando il parametro `y` ad un valore intero: detto `v` tale valore, l'effetto dell'esecuzione del metodo è di incrementare di `v` il valore della variabile istanza `x` di `o`. Vedremo più avanti in questa sezione la sintassi e la semantica della chiamata dei metodi. È chiaro comunque che chiamate diverse del metodo possono essere effettuate istanziando in modi diversi il parametro del metodo stesso.

[15] In Java, la presenza di variabili di tipo `static` consente in generale di modificare anche lo stato globale.

5.6.5 Invocazione dei metodi istanza

Nel caso in cui il metodo invocato non calcoli un valore come risultato, la chiamata di un metodo è un comando, secondo la seguente estensione della categoria sintattica `Com`.

```
Com ::=  Ide.Ide(Exp);
```

Nella chiamata `o.m(E)`, `o` deve essere l'identificatore di un oggetto creato in precedenza (è l'oggetto di riferimento `this`) e `m` deve essere l'identificatore di un metodo della classe di `o`. L'espressione `E` costituisce l'istanziazione del parametro di `m` (ricordiamo che, per semplicità, nel nostro frammento consideriamo metodi che hanno esattamente un parametro).

Riconsideriamo la definizione della classe `Incr` dell'esempio 5.6.

```
class Incr
    {
        public int x;
        public void inc (int y)
           {
             this.x = this.x+y;
           }
    }
```

Intuitivamente, l'effetto di tale chiamata dovrebbe essere di incrementare di `8` il valore della variabile istanza `x` dell'oggetto `o`. Infatti, l'unico comando del metodo aumenta il valore di `this.x` del valore di `y`, il parametro, che in questo caso è stato istanziato con `8`. Dunque nell'esempio che stiamo considerando, `y` è il parametro formale di `m` e `8` è il parametro attuale nella chiamata `obj.inc(8);`.

Vediamo come, da un punto di vista semantico, si trattano le chiamate dei metodi in modo da ottenere l'effetto intuitivo come delineato nell'esempio. Consideriamo dunque una generica chiamata del tipo:

`o.m(E);`

Le azioni da intraprendere sono le seguenti:

(i) reperire nello stato tutte le componenti necessarie alla valutazione della chiamata, ovvero la coppia $(\mathtt{x}, \mathtt{B})$ associata al metodo `m` nell'ambiente delle classi;

(ii) valutare nello stato corrente l'espressione `E` in modo da ottenere il valore del parametro attuale da associare al parametro formale `x`;

(iii) reperire l'indirizzo dello heap dell'oggetto di riferimento del metodo.

(iv) valutare il corpo `B` a partire da uno stato in cui il parametro formale `x` è stato opportunamente legato al valore del parametro attuale e l'identificatore speciale `this` è legato all'indirizzo dell oggetto `o`.

Il fatto che la chiamata di un metodo non possa modificare altro che le variabili istanza dell'oggetto coinvolto, si riflette dal punto di vista semantico nella struttura della seconda componente σ dello stato in cui viene eseguito il corpo `B`. Ad esempio, in corrispondenza di una chiamata del tipo `o.m(E);`, lo stato è

$$\{\mathtt{x} \mapsto \mathtt{v}, \mathtt{this} \mapsto \ell\}.\Omega \qquad (\dagger)$$

dove ℓ sta ad indicare il riferimento associato all'oggetto `o` nello stato al momento della chiamata.

Uno stato come (†) riflette in maniera esplicita il fatto che durante l'esecuzione di un metodo un oggetto è in grado di accedere esclusivamente al suo proprio stato interno, oltre che al parametro del metodo.

Formalmente otteniamo la seguente regola per la chiamata dei metodi, che estende il sistema $\rightarrow_{com}$.

$$\frac{\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi') \qquad \rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad \rho_m(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \qquad \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \omega[{}^{\mathtt{v}}/_{\mathtt{x}}, {}^{\sigma(\mathtt{o})}/_{\mathtt{this}}].\Omega \qquad \langle \mathtt{B}, \langle \rho_c, \sigma', \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta' \rangle}{\langle \mathtt{o.m(E)}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma, \zeta' \rangle} \quad (com_{InstCall})$$

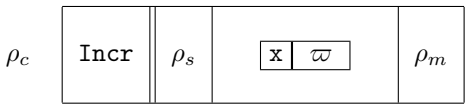
È importante osservare che l'invocazione di un metodo calcola in ogni caso uno stato in cui l'unica componenete che risulta modificata è lo heap, mentre la seconda componente , ovvero lo stack di frame σ, viene restituito inalterato. Questo risulta evidente dalla precedente regola semantica. Ciò significa che la chiamata di un metodo può modificare solo variabili istanza di oggetti coinvolti nella chiamata.

Esempio 5.7 Consideriamo di nuovo la classe dell'esempio 5.6, che in realtà deve essere riscritta come segue per rappresentare i riferimenti alle variabili istanza:

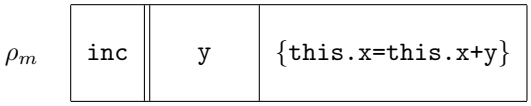
```
class Incr
    {
        public int x;
        public void inc (int y)
           {
             this.x = this.x+y;
           }
    }
```

Consideriamo una chiamata del tipo `obj.inc(10);` a partire dallo stato $\langle \rho_c, \varphi.\Omega, \zeta \rangle$ in cui:

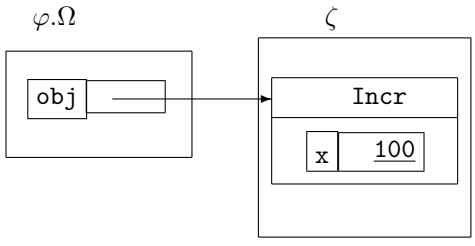
- l'ambiente delle classi ρ_c è il seguente:



  dove $\rho_s = \omega$ mentre ρ_m è:



- lo stato $\varphi.\Omega$ e lo heap ζ sono rappresentati dalla seguente figura:



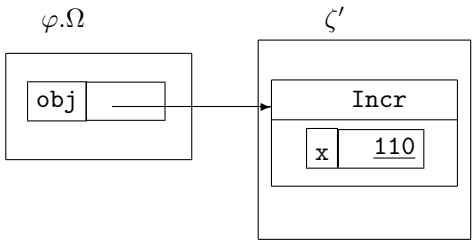
Utilizzando la regola (com_{call}), è facile convincersi che si può costruire la derivazione

$\langle \mathtt{obj.inc(10);}, \langle \rho_c, \sigma, \zeta \rangle \rangle$

$\rightarrow_{com}$

$\langle \sigma, \zeta' \rangle$

dove il nuovo heap ζ' è rappresentato come segue:



Anche i metodi d'istanza possono calcolare un valore (in Java e in tutti i linguaggi di programmazione). In questo caso la regola semantica che possiamo pensare di definire, combinando le due regole semantiche del invocazione di metodo statico che calcola un valore e quella di metodo d'istanza che non calcola un valore, potrebbe essere la seguente:

$$\frac{\begin{array}{c}\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi') \qquad \rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad \rho_m(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \\ \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \omega[{}^{\mathtt{v}}/_{\mathtt{x}}, {}^{\sigma(\mathtt{o})}/_{\mathtt{this}}, {}^{\varpi}/_{\mathtt{retval}}].\Omega \\ \langle \mathtt{B}, \langle \rho_c, \sigma', \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta' \rangle \end{array}}{\langle \mathtt{o.m(E)}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \sigma''(\mathtt{retval})} \quad (exp_{InstCallVal})$$

oppure la seguente:

$$\frac{\begin{array}{c}\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi') \qquad \rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad \rho_m(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \\ \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \omega[{}^{\mathtt{v}}/_{\mathtt{x}}, {}^{\sigma(\mathtt{o})}/_{\mathtt{this}}, {}^{\varpi}/_{\mathtt{retval}}].\Omega \\ \langle \mathtt{B}, \langle \rho_c, \sigma', \zeta \rangle \rangle \rightarrow_{com} \langle \sigma'', \zeta' \rangle \end{array}}{\langle \mathtt{o.m(E)}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma''(\mathtt{retval}), \zeta' \rangle} \quad (exp_{InstCallVal2})$$

Il lettore è invitato a riflettere sulle due soluzioni, formandosi, prima di procedere nella lettura, una propria opinione su la correttezza di tali regole. A tale scopo è senza dubbio utile riconsiderare le regole per le espressioni definite finora.

Il problema che nasce dall'introduzione di tali metodi (cioè i metodi d'istanza che calcolano un valore) è in realtà molto piu complesso di quello che potrebbe apparire a prima vista. Infatti nessuna delle due regole definite sopra è in realtà corretta se unita all'insieme delle regole date finora. La prima rende inutili le modifiche apportate nel corpo del metodo sullo heap, dal momento che lo heap risultante non viene utilizzato in alcun modo. La seconda non fa parte del sistema di transizioni $\rightarrow_{exp}$, che prevede come configurazioni terminali solo valori e non coppie $\langle \mathtt{v}, \zeta \rangle$. Infatti, la soluzione data finora era corretta nell'ipotesi che un'espressione non potesse mai modificare lo stato, in nessuna sua componente. Nel momento in cui abbiamo introdotto i metodi d'istanza che possono modificare lo heap e contemporaneamente calcolare un valore, tale ipotesi non risulta più vera. A questo punto è necessario prevedere che le espressioni calcolino oltre al valore anche uno heap, che concorrerà a formare lo stato per proseguire la valutazione in accordo alle nuove regole. È necessario quindi prevedere la ridefinizione dell'intero sistema di transizioni per le espressioni ed i comandi, definendo come semantica delle espressioni una coppia $\langle \mathtt{v}, \zeta \rangle$. Il nuovo sistema di transizioni per le espressioni è allora $S_{exp} = \langle \Gamma_{exp}, T_{exp}, \rightarrow_{exp} \rangle$, in cui:

$$
\begin{array}{rcl}
\Gamma_{exp} & = & \{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \mid E \in Exp, \rho_c \in \mathtt{ClassEnv}, \sigma \in \Sigma, \zeta \in \mathtt{Heap}\} \cup \\
 & & \{\langle \underline{\mathtt{n}}, \zeta\rangle\rangle \mid \underline{\mathtt{n}} \in I\!N, \zeta \in \mathtt{Heap}\} \\
\mathtt{T}_{exp} & = & \{\langle \underline{\mathtt{n}}, \zeta\rangle\rangle \mid \underline{\mathtt{n}} \in I\!N, \zeta \in \mathtt{Heap}\}
\end{array}
$$

mentre il sistema di transizioni per i comandi rimane invariato per quanto riguarda le configurazioni mentre sono variate solo le regole. Riportiamo di seguito le nuove e definitive regole per $\rightarrow_{exp}$:

$$\langle \mathtt{n}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta\rangle \qquad (exp_n)$$

$$\frac{\sigma(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta\rangle} \qquad (exp_{ide})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle \qquad \langle \mathtt{E}', \langle \rho_c, \sigma, \zeta''\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}', \zeta'''\rangle \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\langle \mathtt{E}{+}\mathtt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{m}}, \zeta'''\rangle} \qquad (exp_+)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle \qquad \langle \mathtt{E}', \langle \rho_c, \sigma, \zeta''\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}', \zeta'''\rangle \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\langle \mathtt{E}{*}\mathtt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{m}}, \zeta'''\rangle} \qquad (exp_*)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle \qquad \langle \mathtt{E}', \langle \rho_c, \sigma, \zeta''\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}', \zeta'''\rangle \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\langle \mathtt{E}\text{-}\mathtt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{m}}, \zeta'''\rangle} \qquad (exp_-)$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle \qquad \langle \mathtt{E}', \langle \rho_c, \sigma, \zeta''\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}', \zeta'''\rangle \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ div\ \underline{\mathtt{n}}'}{\langle \mathtt{E}\ /\ \mathtt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{m}}, \zeta'''\rangle} \qquad (exp_{div})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle \qquad \langle \mathtt{E}', \langle \rho_c, \sigma\zeta''\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}', \zeta'''\rangle \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ mod\ \underline{\mathtt{n}}'}{\langle \mathtt{E}\ \%\ \mathtt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{m}}, \zeta'''\rangle} \qquad (exp_{mod})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle}{\langle (\mathtt{E}), \langle \rho_c, \sigma\zeta\rangle\rangle \rightarrow_{exp} \langle \underline{\mathtt{n}}, \zeta''\rangle} \qquad (exp_{()})$$

$$\frac{\begin{array}{c}\rho_c(\mathtt{c}) = (\rho_s, \varphi, \rho_m) \qquad \rho_s(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \qquad \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta''\rangle \\ \sigma'' = \omega[^{\mathtt{v}}/_{\mathtt{x}}, ^{\varpi}/_{\mathtt{retval}}].\Omega \qquad \langle \mathtt{B}, \langle \rho_c, \sigma'', \zeta''\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle\end{array}}{\langle \mathtt{c.m(E)}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma'(\mathtt{retval}), \zeta'\rangle} \qquad (exp_{StatCall})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma\rangle\rangle \rightarrow_{exp} \mathtt{v}}{\langle \mathtt{return\ E}, \langle \rho_c, \sigma\rangle\rangle \rightarrow_{com} \sigma[^{\mathtt{v}}/_{\mathtt{retval}}]} \qquad (exp_{return})$$

$$\frac{\begin{array}{c}\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi') \qquad \rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad \rho_m(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \\ \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta''\rangle \qquad \sigma'' = \omega[^{\mathtt{v}}/_{\mathtt{x}}, ^{\sigma(\mathtt{o})}/_{\mathtt{this}}, ^{\varpi}/_{\mathtt{retval}}].\Omega \\ \langle \mathtt{B}, \langle \rho_c, \sigma'', \zeta''\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle\end{array}}{\langle \mathtt{o.m(E)}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma'(\mathtt{retval}), \zeta'\rangle} \qquad (exp_{InstCallVal2})$$

Riportiamo inoltre le regole semantiche per $\rightarrow_{dec}$ aggiornate con lo stato completo ed eventuali valutazioni di espressioni:

$$\frac{\sigma = \varphi.\sigma'' \qquad \sigma' = \varphi[^{\varpi}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x;}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{dec} \langle \sigma', \zeta \rangle} \quad (dec_{var-1})$$

$$\frac{\sigma = \varphi.\sigma'' \qquad \langle \mathtt{E}, \langle \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta' \rangle \qquad \sigma' = \varphi[^{\mathtt{v}}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x = E;}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{dec} \langle \sigma', \zeta' \rangle} \quad (dec_{var-2})$$

Infine le nuove e definitive regole semantiche per $\rightarrow_{com}$ sono le seguenti:

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta' \rangle}{\langle \mathtt{x=E;}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma[^{\mathtt{v}}/_{\mathtt{x}}], \zeta' \rangle} \quad (com_{=})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle tt, \zeta'' \rangle \qquad \langle \mathtt{C1}, \langle \rho_c, \sigma, \zeta'' \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle} \quad (com_{if-tt})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle ff, \zeta'' \rangle \qquad \langle \mathtt{C2}, \langle \rho_c, \sigma, \zeta'' \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle} \quad (com_{if-ff})$$

$$\frac{\begin{array}{c}\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle tt, \zeta'' \rangle \qquad \langle \mathtt{C}, \langle \rho_c, \sigma, \zeta'' \rangle\rangle \rightarrow_{com} \langle \sigma'', \zeta''' \rangle \\ \langle \mathtt{while\ (E)\ \ C}, \langle , \rho_c, \sigma'', \zeta''' \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle\end{array}}{\langle \mathtt{while\ (E)\ \ C}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle} \quad (com_{while-tt})$$

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle ff, \zeta' \rangle}{\langle \mathtt{while\ (E)\ \ C}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle} \quad (com_{while-ff})$$

$$\frac{\begin{array}{c}\rho_c(\mathtt{c}) = (\rho_s, \varphi, \rho_m) \qquad \rho_s(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \qquad \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta'' \rangle \\ \sigma' = \omega[^{\mathtt{v}}/_{\mathtt{x}}].\Omega \qquad \langle \mathtt{B}, \langle \rho_c, \sigma', \zeta'' \rangle\rangle \rightarrow_{com} \langle \sigma'', \zeta' \rangle\end{array}}{\langle \mathtt{c.m(E)}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma, \zeta' \rangle} \quad (com_{call})$$

$$\frac{\begin{array}{c}\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi') \qquad \rho_c(\mathtt{c}) = (\varphi, \rho_m) \qquad \rho_m(\mathtt{m}) = (\mathtt{x}, \mathtt{B}) \\ \langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \langle \mathtt{v}, \zeta'' \rangle \qquad \sigma' = \omega[^{\mathtt{v}}/_{\mathtt{x}}, ^{\sigma(\mathtt{o})}/_{\mathtt{this}}].\Omega \\ \langle \mathtt{B}, \langle \rho_c, \sigma'', \zeta'' \rangle\rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle\end{array}}{\langle \mathtt{o.m(E)}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma, \zeta' \rangle} \quad (com_{InstCall})$$

Per completezza aggiungiamo, infine, anche le regole semantiche per il blocco e le liste di statements (comandi e dichiarazioni) anche se tali regole non risultano modificate in quanto non contengono valutazioni di espressioni.

$$\frac{\langle \texttt{Slist}, \langle \rho_c, \omega.\sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \varphi.\sigma', \zeta' \rangle}{\langle \{\texttt{Slist}\}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle} \quad (com_{Block})$$

$$\frac{\texttt{S} \in \texttt{Decl} \qquad \langle \texttt{S}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{dec} \langle \sigma', \zeta' \rangle}{\langle \texttt{S}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle} \quad (slist_{dec})$$

$$\frac{\texttt{S} \in \texttt{Com} \qquad \langle \texttt{S}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{com} \langle \sigma', \zeta' \rangle}{\langle \texttt{S}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle} \quad (slist_{com})$$

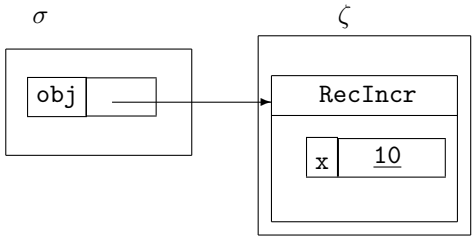
$$\frac{\langle \texttt{S}, \langle \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma'', \zeta'' \rangle \qquad \langle \texttt{Slist}, \langle \rho_c, \sigma'', \zeta'' \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle}{\langle \texttt{S Slist}, \langle \rho_c, \sigma, \zeta \rangle \rangle \rightarrow_{slist} \langle \sigma', \zeta' \rangle} \quad (slist_{list})$$

5.7 Metodi ricorsivi

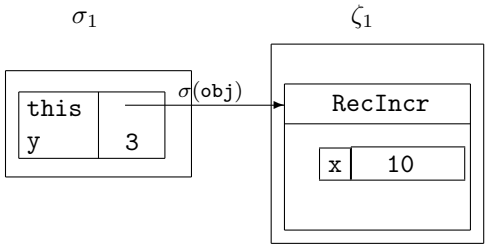
Il linguaggio Java, così come il frammento da noi utilizzato, consente la definizione di metodi *ricorsivi*. Un metodo è ricorsivo se, nella sua definizione, utilizza chiamate a sé stesso. Vediamo brevemente come la semantica operazionale consenta di trattare correttamente anche questa caratteristica del linguaggio. A questo scopo, consideriamo quale esempio la seguente classe.

```
class RecIncr
    {
        public int x;
        public void inc (int y)
           {
             if (y<=1)
                this.x = this.x+1;
             else
                {
                    this.x=this.x+1;
                    this.inc (y-1);
                }
           }
    }
```

Supponiamo che `obj` identifichi un oggetto della classe sopra descritta e supponiamo di eseguire la chiamata `obj.inc(3)` a partire da uno stato in cui la pila di frame e lo heap contengono, tra le altre, le associazioni descritte nella seguente figura.

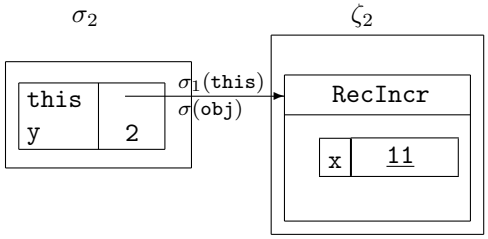


Vediamo come evolve la chiamata `obj.incr(3)`, mostrando graficamente solo le componenti significative dello stato. Secondo la regola (com_{call}), il corpo del metodo viene eseguito nello stato in cui le componenti "pila di frame" e heap sono le seguenti:



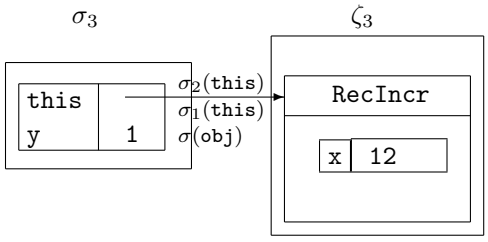
Per chiarezza, abbiamo indicato esplicitamente il valore $\sigma(\texttt{obj})$ del riferimento associato all'identificatore speciale `this`; si noti inoltre che $\zeta_1 = \zeta$.

Poiché nello stato corrente il valore di `y` è strettamente maggiore di 1, viene eseguito il ramo `else` del corpo del metodo: dopo l'assegnamento `this.x=this.x+1` viene eseguita (ancora grazie alla regola (com_{call})) la chiamata ricorsiva `this.inc(y-1)` nello stato seguente:

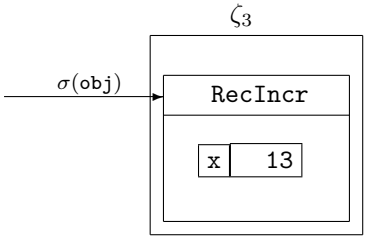


Si noti che la chiamata ricorsiva viene eseguita a partire da un *nuovo* frame che contiene l'associazione per il parametro attuale della chiamata ricorsiva stessa, oltre che l'associazione per l'identificatore speciale `this`: quest'ultimo è associato al riferimento $\sigma_1(\texttt{this})$, che equivale, a sua volta a $\sigma(\texttt{obj})$. Dunque, ad ogni chiamata ricorsiva l'identificatore `this` presente nell'unico frame dello stato è sempre associato all'oggetto di partenza.

L'esecuzione della chiamata ricorsiva porta ad un'ulteriore chiamata ricorsiva `this.inc(y-1)` che avviene nello stato:



Poiché in questo stato il valore di `y` è uguale a 1, viene eseguito il solo comando `this.x = this.x +1`. Inoltre la ricorsione ha termine e viene restituito lo heap rappresentato dalla seguente figura:

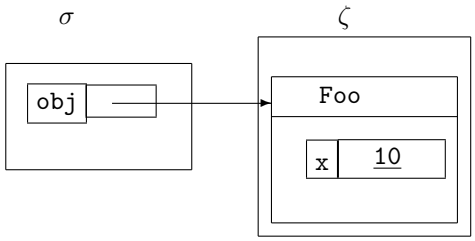


È questo lo heap che viene restituito, a loro volta, dalle chiamate ricorsive del metodo `inc` e dalla stessa chiamata iniziale `obj.this(3);`. Come ci potevamo aspettare, l'effetto di quest'ultima sullo stato complessivo si riflette nel fatto che il valore della variabile istanza `x` di `obj` è stato incrementato di 3.

Consideriamo adesso un ulteriore esempio che mette meglio in luce come gli effetti delle chiamate ricorsive si ripercuotano a ritroso dalla chiamata più annidata a quella iniziale.

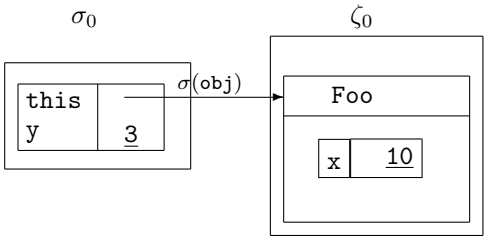
```
class Foo
    {
        public int x;
        public void fie (int y)
           {
             if (y<=1)
                this.x = this.x+1;
             else
                this.fie (y-1);

             this.x = 2*this.x;
           }
    }
```

Come nell'esempio precedente, eseguiamo la chiamata `obj.fie(3)`, assumendo che `obj` identifichi un oggetto della classe `Foo` e che lo stato iniziale sia quello descritto dalla figura seguente.

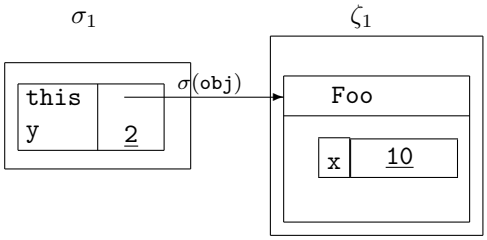


È facile convincersi che la sequenza degli stati in cui vengono eseguite le chiamate ricorsive annidate è la seguente:

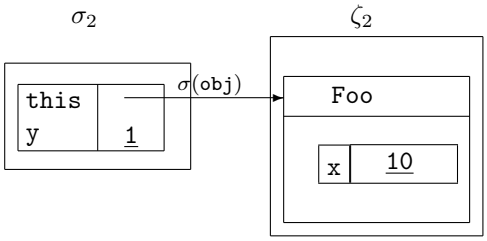
Chiamata iniziale



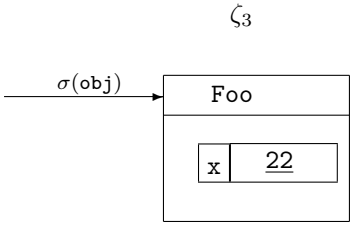
1^a chiamata ricorsiva



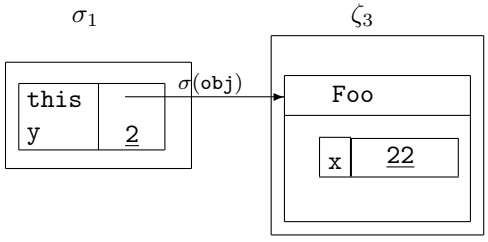
2^a chiamata ricorsiva



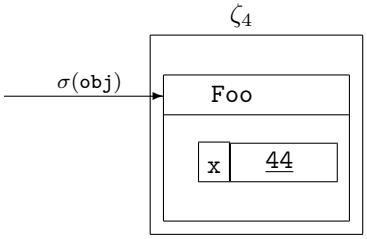
Poiché in quest'ultimo stato il valore di `y` è 1, viene eseguito il primo ramo del comando condizionale (l'assegnamento `this.x = this.x+1;`) e, successivamente, il comando `this.x = 2*this.x;`, con il quale ha termine la 2^a chiamata ricorsiva. Lo heap restituito è il seguente:



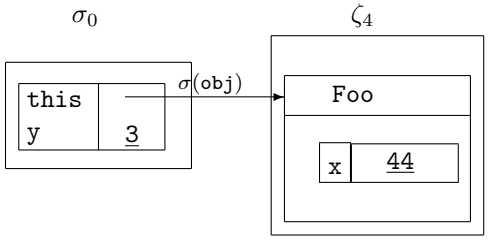
A questo punto si prosegue con l'esecuzione dell'ultimo comando (`this.x=2*this.x;`) della prima chiamata ricorsiva, a partire dallo stato



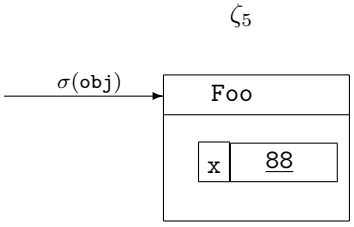
Si osservi come la componente heap dello stato rifletta le modifiche apportate dalla chiamata ricorsiva annidata. Con l'esecuzione dell'assegnamento `this.x=2*this.x;` ha termine anche la 1^a chiamata ricorsiva, che restituisce lo heap



Si prosegue infine con l'esecuzione dell'ultimo comando (`this.x=2*this.x;`) della chiamata iniziale, a partire dallo stato



che termina restituendo lo heap



Lo stato finale calcolato dall'invocazione `obj.fie(3)` è allora il seguente