+/- Java
Semantica Operazionale

Università degli Studi dell'Aquila

Facoltà di Scienze Matematiche,
Fisiche e Naturali

Corso di Laurea in Informatica

M. Autili, P. Inverardi

3 dicembre 2010

Prefazione

Queste note affrontano lo studio della semantica operazionale di un semplice linguaggio di programmazione che contiene i principali costrutti del linguaggio Java, ma con delle restrizioni che ne rendono più semplice la trattazione. Esse nascono da una profonda rivisitazione della loro omonima prima versione e sono pertanto il frutto di uno sforzo congiunto di tutti gli autori e revisori coinvolti:

- Marco Autili
- Antinisca Di Marco
- Benedetto Intrigila
- Paola Inverardi
- Fabio Mancinelli
- Monica Nesi
- Patrizio Pelliccione

In questa versione tutti i contenuti sono presentati in modo che il lettore abbia, in ogni momento, un collegamento diretto con il mondo Java. In questa direzione, la terminologia utilizzata per l'esposizione dei contenuti fa riferimento alle specifiche ufficiali del Linguaggio Java e della Java Virtual Machine. In particolare, questa versione rivisita leggermente anche quella immediatamente precedente del 10 gennaio 2010 rispetto alla quale alcuni contenuti sono stati spostati per migliorare l'ordine di esposizione, alcune funzioni sono state ridefinite e alcuni errori sono stati corretti.

Si invitano comunque i lettori a leggere con criticità queste note in modo da rilevare ulteriori errori sia tecnici sia di battitura. Si ringraziano in anticipo tutti coloro che ne segnaleranno la presenza. Naturalmente, sono altrettanto benvenuti commenti in merito alla qualità dei contenuti, la loro presentazione e la conseguente facilità di lettura.

Si ringraziano in particolare gli autori delle note "Semantica Operazionale di R. Barbuti, P. Mancarella e F. Montangero" e delle note "Elementi di Semantica Operazionale - Appunti per gli studenti di Fondamenti di Programmazione - Corso di Laurea in Informatica, Universita' di Pisa - A.A. 2004/05 di R. Barbuti, P. Manacarella e F. Turini" a cui queste dispense si ispirano per i concetti alla base della definizione dello stato.

1 Introduzione

In queste note affronteremo lo studio della semantica operazionale di un frammento di linguaggio di programmazione orientato agli oggetti. Tale linguaggio contiene i principali costrutti del linguaggio Java, ma con delle semplificazioni che ne rendono più semplice la trattazione. Ci riferiremo al linguaggio che tratteremo con il termine +/-Java mantenendo così traccia della sua origine.

Faremo un breve introduzione a Java con il solo scopo di introdurre i concetti di base che ci permetteranno di collocare la nostra semantica operazionale nel mondo Java. Pertanto, non descriveremo in dettaglio la piattaforma Java ma assumeremo che il lettore sia familiare con i concetti fondamentali del linguaggio e della programmazione orientata ad oggetti. Assumeremo inoltre che il lettore abbia familiari i concetti introdotti delle note *Semantica Operazionale di R. Barbuti, P. Mancarella e F. Montangero* (da pag. 1 a pag. 39) oppure nella rivisitazione *Elementi di Semantica Operazionale - Appunti per gli studenti di Fondamenti di Programmazione - Corso di Laurea in Informatica, Universita' di Pisa - A.A. 2004/05 di R. Barbuti, P. Manacarella e F. Turini* (da pag. 1 a pag. 48).

2 Il linguaggio Java e la Java Virtual Machine

Nel mondo Java è importante distinguere il *Linguaggio Java* e la *Java Virtual Machine* (di seguito riferita come JVM).

Questa sezione introduce brevemente, ed in modo molto semplificato, i concetti di base per comprendere la relazione che c'è tra il linguaggio Java e la JVM. Sebbene non indispensabili per la comprensione di queste note, è utile sapere che le specifiche ufficiali del linguaggio e della JVM possono essere trovate sul sito ufficiale della SUN (http://java.sun.com) sotto i nomi "*The Java^{TM} Language Specification*" e "*The Java^{TM} Virtual Machine Specification*", rispettivamente.

2.1 Il linguaggio Java

Il linguaggio Java nasce da un progetto di Sun Microsystem, "Green", ed è stato ufficializzato nel 23 maggio 1995. Java è un *linguaggio di programmazione orientato agli oggetti* e permette di realizzare applicazioni software in termini di oggetti che interagiscono tra loro.

L'idea alla base della programmazione orientata agli oggetti è quella di utilizzare oggetti (istanziati da classi) per rappresentare le entità reali o as-

tratte di un dominio di interesse. Semplicemente parlando, ogni oggetto è caratterizzato dai suoi **membri**, e cioè dalle variabili (o **campi**) e dalle funzioni (o **metodi**) che incapsula[1]. I metodi possono ad esempio modificare lo stato delle variabili oppure estrarre informazioni da esse. Gli oggetti posso essere di vario tipo e per definire varie tipologie di oggetti si utilizza il concetto di *classe* che rappresenta il modello (la struttura, la forma) dal quale è poi possibile creare istanze di specifici oggetti. Più precisamente una classe è un *tipo di dato astratto* che **modella** un'entità reale o astratta del dominio applicativo di interesse. Un oggetto è un'istanza di una classe.

Per esempio, se il dominio applicativo di interesse è l'anagrafe, un'entita reale da modellare come classe potrebbe essere l'entità persona che ha un nome, un cognome ed in indirizzo. Il seguente codice sorgente Java definisce (parte di) una classe `Persona` e quindi il tipo di dato astratto che modella un persona.

Persona.java

```
class Persona {
  private String nome;
  private String cognome;
  private String indirizzo;

  public Persona(String n, String c, String ind) {
    this.nome = n;
    this.cognome = c;
    this.indirizzo = ind;
  }

  public String get_indirizzo() {
    return this.indirizzo;
  }

  public void set_indirizzo(String indirizzo) {
    this.indirizzo = indirizzo;
  }
  ...
}
```

[1] In particolare, nei linguaggi di programmazione orientati agli oggetti, l'incapsulamento è un meccanismo che fornisce la possibilità di mettere insieme dati e codice per manipolarli, cioè metodi).

La classe `Persona` definisce anche dei metodi (detti *metodi di istanza*) tra cui `String get_indirizzo() {...}` per estrarre il valore della *variabile d'istanza* indirizzo e `void set_indirizzo(String indirizzo) {...}` per modificare l'indirizzo di una persona. In particolare, il metodo `Persona(String n, String c, String ind) {...}`, che non ha specificato il tipo di ritorno ed ha lo stesso nome della classe, è il *costruttore* della classe e viene richiamato una sola volta in fase di creazione di una nuova *istanza* della classe persona. Un'istanza della classe `Persona` potrebbe essere l'oggetto che rappresenta una specifica persona di nome Mario e cognome Rossi che abita in Piazza Duomo - 67100 - L'Aquila.

I programmi scritti in Java possono essere unicamente orientati agli oggetti e quindi tutto il codice deve essere necessariamente incluso in classi. Comunque, Java lavora sia su *tipi di dati astratti*, quali le classi, sia su *tipi di dati primitivi*, quali i numeri interi e booleani. Ciononostante, Java è considerato un linguaggio ad oggetti *puro*, nel quale cioè gli oggetti sono gli elementi di base del linguaggio, anzichè essere costruiti partendo da costrutti di più basso livello.

Per scrivere il codice sorgente di programmi Java sintatticamente corretti si utilizza la grammatica di Java. Il codice sorgente viene memorizzato in file con formato testo semplice aventi estensione *.java*. Per esempio, per la classe `Persona` avremo un file con nome *Persona.java*[2]. In queste note presenteremo una grammatica per il nostro +/-Java che permetterà la scrittura di un sottoinsieme di codici sorgenti Java sintatticamente corretti.

Come meglio spiegato nella prossima sezione, Java permette la realizzazione di applicazioni indipendenti dall'architettura del calcolatore su cui verranno poi eseguite e quindi indipendenti dalle differenti istruzioni macchina delle differenti CPU. In questa direzione, Java promuove la *portabilità* del codice seguendo la filosofia "*scrivi una volta ... esegui ovunque*"[3]. Per ottenere indipendenza e quindi portabilità, Java si basa su una *standardizzazione "virtuale"* del processore e delle sue istruzioni di programmazione che è realizzata dalla JVM. Riferendosi a questo processore "campione", che non è quello reale/fisico, il codice sorgente Java è tradotto (compilato) in un codice noto come *bytecode* (codice a byte). Quindi, il bytecode è un codice che contiene tutte le informazioni del codice sorgente tradotte però in istruzioni (e strutture dati) interpretabili dalla JVM.

[2]In realtà si possono definire più classi in un file.

[3]Tutta la documentazione sulle versioni ufficiali di Java e il software di supporto allo sviluppo in Java si può trovare sul sito ufficiale della SUN (http://java.sun.com).

2.2 La Java Virtual Machine

La JVM è stata progettata per supportare il linguaggio di programmazione Java. La Sun rilascia sia la JVM che un *compilatore* (chiamato *javac*) che permette di tradurre codici sorgenti scritti in Java nell'insieme di istruzioni della JVM e quindi bytecode.

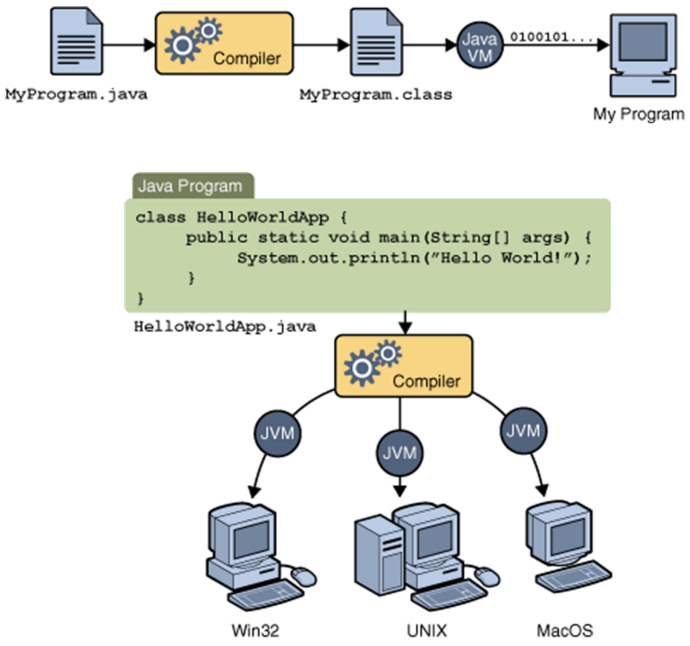

Figura 1: Per mezzo della JVM e del bytecode una stessa applicazione può girare su differenti piattaforme

Sebbene il nome può trarre in inganno, la JVM è una macchina di computazione reale capace di eseguire un insieme di istruzioni e capace di manipolare aree di memoria. Comunque, a differenza di altri linguaggi, quali il C++, il compilatore Java non produce codice dipendente dalla macchina (ed in particolare dal processore reale) e dal sistema operativo, ma produce sempre bytecode. Poi, la JVM interpreta questo bytecode ed esegue le istruzioni in esso contenute. La JVM non conosce nulla del linguaggio Java e della sua sintassi, e quindi non conosce nulla del codice sorgente. La JVM conosce solo il bytecode e quindi il formato dei file *.class* che lo contengono. Questo implica che il compilatore *javac* riporta (traduce) tutte le informazioni contenute nel sorgente di un programma Java nel formato bytecode. Per esempio, il seguente codice è il bytecode relativo alla classe `Persona` ottenuto compilando il file *Persona.java* con il compilatore *javac*[4]. Il codice è stato mostrato solo

[4]In realtà il bytecode sarebbe più complesso. Quello mostrato è stato ottenuto con il

a titolo esemplificativo in quanto esula gli scopi di queste note la trattazione del bytecode. Comunque, il lettore interessato può confrontare le istruzioni bytecode con il codice sorgente Java e trarre facilmente utili conclusioni.

Persona.class

```
Compiled from "Persona.java" class prove.persona.Persona extends
java.lang.Object{ public prove.persona.Persona(java.lang.String,
java.lang.String, java.lang.String);
  Code:
   0:   aload_0
   1:   invokespecial   #12; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   aload_1
   6:   putfield        #15; //Field nome:Ljava/lang/String;
   9:   aload_0
   10:  aload_2
   11:  putfield        #17; //Field cognome:Ljava/lang/String;
   14:  aload_0
   15:  aload_3
   16:  putfield        #19; //Field indirizzo:Ljava/lang/String;
   19:  return

public java.lang.String get_indirizzo();
  Code:
   0:   aload_0
   1:   getfield        #19; //Field indirizzo:Ljava/lang/String;
   4:   areturn

public void set_indirizzo(java.lang.String);
  Code:
   0:   aload_0
   1:   aload_1
   2:   putfield        #19; //Field indirizzo:Ljava/lang/String;
   5:   return
}
```

2.2.1 *Scrivi una volta ... esegui ovunque*

La Figura 1, tratta dal sito ufficiale della SUN, riassume graficamente quanto detto finora. In Java tutto il codice sorgente di una applicazione è scritto nel rispetto della sintassi del linguaggio Java in file con estensione *.java*. Il compilatore Java (*javac*) compila tutti questi file in file con estensione *.class*[5]. Ogni file *.class* non contiene codice nativo di nessun particolare processore ma contiene solo il linguaggio macchina interpretabile dalla JVM (cioè istruzioni bytecode). Infine, un programma di nome *java* lancia una istanza della JVM ed esegue l'applicazione su di essa. In conclusione, siccome ogni sistema operativo ha la propria JVM, gli stessi file *.class* possono essere eseguiti su

programma *javap* (anch'esso fornito dalla SUN) che permette di visualizzare un file *.class* in modo più leggibile.

[5]Il compilatore produce un file *.class* per ogni classe definita anche nel caso in cui più classi sono definite nello stesso file *.java*.

differenti sistemi operativi quali Microsoft Window, Solaris OS, Linux oppure MAC OS.

3 Semantica statica e dinamica

Come già introdotto, queste note trattano la semantica operazionale di un frammento del linguaggio Java. A tal proposito, questa sezione spiega la differenza tra una *semantica statica* e una *semantica dinamica* facendo riferimento al mondo Java come brevemente descritto nelle sezioni precedenti. Prima di fare ciò può essere utile fare la seguente considerazione.

Richiamando quanto detto in Sezione 2.1, la JVM, come il linguaggio Java, lavora su *tipi primitivi*, tra cui il tipo intero e il tipo boolean, e su *tipi di dati astratti*, quali le classi. Per il momento, la cosa importante da sottolineare è che la JVM assume che il controllo di correttezza dei tipi sia stato fatto a priori, e quindi *staticamente* prima dell'esecuzione del programma. Infatti, questo e molti altri controlli sono tipicamente effettuati durante l'analisi statica fatta compilatore prima dell'esecuzione e non devono quindi essere effettuati dalla JVM.

Di seguito ci riferiremo ai controlli effettuati nella fase di compilazione con il termine *controlli statici*. Inoltre, chiameremo *semantica statica* una semantica che formalizza il comportamento operativo del compilatore *javac* durante la fase di traduzione dei sorgenti *in file .java* in istruzioni bytecode *in file .class*. Per esempio, una semantica statica potrebbe riguardare la verifica di proprietà da parte del compilatore come inferenza e controllo dei tipi, utilizzo di variabili non dichiarate, ridichiarazioni di variabili, etc. In generale quindi, una semantica statica può riguardare tutte le fasi dopo la scrittura del programma e prima della sua esecuzione da parte della JVM.

Invece, **la semantica operazionale del nostro +/-Java ha come scopo quello di formalizzare un comportamento operativo consistente con quello della JVM (quindi a run-time) e sarà perciò una *semantica dinamica.*** Come vedremo, la semantica che stiamo per presentare prenderà in considerazione aspetti come allocazione dinamica della memoria, controlli a run-time e creazione delle strutture dati necessarie per l'esecuzione di un programma.

La semantica sarà definita usando il formalismo dei sistemi di transizione. Verrà quindi definita una macchina formale astratta utilizzando un sistema di transizione che, a fronte di un programma Java, passa di configurazione in configurazione mostrando un comportamento operativo consistente con quello della JVM (o meglio una sua semplificazione). Vale la pena notare che, siccome la JVM legge il programma in file *.class* ed esegue le istruzioni

bytecode in essi contenute, avremmo dovuto definire la nostra macchina formale astratta per il linguaggio bytecode. Siccome questo renderebbe molto complicata la sua presentazione, la nostra macchina formale astratta lavorerà su codice sorgente.

4 Grammatica

Come già detto, in queste note tratteremo +/-Java, un linguaggio che contiene i principali costrutti del linguaggio Java ma con delle semplificazioni che ne rendono più semplice la trattazione. Nella Sezione 4.1 si elencano le principali restrizioni che il nostro +/-Java impone al linguaggio Java e nella Sezione 4.2 si riporta la grammatica che permette di scrivere solo programmi corretti in +/-Java.

4.1 Semplificazioni del linguaggio

- Le dichiarazioni di classe non hanno modificatori di accesso e sono visibili dalle altre classi dichiarate.
- Non è possibile dichiarare classi annidate.
- Non è possibile dichiarare sottoclassi.
- Struttura rigida dei programmi: prima le dichiarazioni di classe ed infine una dichiarazione standard della classe `Program` contenente il solo metodo `main` (entry point del programma +/-Java).
- Struttura rigida dei blocchi: prima le dichiarazioni di variabili locali ed in seguito la sequenza dei comandi.
- Struttura rigida del corpo dei metodi: prima le dichiarazioni di variabili locali, in seguito la sequenza dei comandi ed infine il costrutto `return` (qualora il metodo deve ritornare un valore).
- Le variabili di una classe sono tutte `private` e non possono essere inizializzate contestualmente alla loro dichiarazione.
- I metodi definiti dentro una classe sono tutti `public`.
- Restrizione dei tipi basici a `int` e `boolean`.
- Accesso alle variabili di una classe esclusivamente mediante il costrutto `this.IDE`.

- Un metodo di una classe può accedere a un altro metodo della stessa classe (oppure se stesso, se ricorsivo) esclusivamente mediante il costrutto `this.METH_IDE`.

4.2 Definizione della grammatica

```
PROGRAM ::= CLASS_DECL_LIST
            public class Program {
              public static void main(String[] IDE) METH_BODY_NO_RETURN
            }

CLASS_DECL_LIST ::= CLASS_DECL CLASS_DECL_LIST |
                    ε

CLASS_DECL ::= class CLASS_IDE { CLASS_MEMBER_DECL_LIST }

CLASS_MEMBER_DECL_LIST ::= CLASS_MEMBER_DECL CLASS_MEMBER_DECL_LIST|
                           ε

CLASS_MEMBER_DECL ::= FIELD_VAR_DECL |
                      METH_DECL |
                      CONSTRUCTOR_DECL

FIELD_VAR_DECL ::= private TYPE_NAME IDE;

TYPE_NAME ::= PRIMITIVE_TYPE_NAME |
              CLASS_IDE

PRIMITIVE_TYPE_NAME ::= int |
                        boolean

CONSTRUCTOR_DECL ::= public CLASS_IDE(PARAM_DECL) METH_BODY_NO_RETURN

METH_DECL ::= public TYPE_NAME METH_IDE(PARAM_DECL) METH_BODY |
              public void METH_IDE(PARAM_DECL) METH_BODY_NO_RETURN

METH_BODY ::= { VAR_DECL_LIST STATEMENT return EXP; }

METH_BODY_NO_RETURN ::= { VAR_DECL_LIST STATEMENT }

PARAM_DECL ::= TYPE_NAME IDE |
               ε
```

```
VAR_DECL_LIST ::= VAR_DECL VAR_DECL_LIST |
                  ε

VAR_DECL ::= TYPE_NAME IDE = EXP; |
             TYPE_NAME IDE;

STATEMENT ::= BASIC_STATEMENT STATEMENT |
              ε

BASIC_STATEMENT ::= IDE = EXP; |
                    this.IDE = EXP; |
                    BLOCK |
                    if(EXP) BASIC_STATEMENT else BASIC_STATEMENT |
                    while(EXP) BASIC_STATEMENT |
                    EXP_STATEMENT; |
                    ;

BLOCK ::= { VAR_DECL_LIST STATEMENT }

EXP ::= CONST_VAL |
        IDE |
        this.IDE |
        EXP OP EXP |
        UNARYOP EXP |
        (EXP) |
        EXP_STATEMENT

EXP_STATEMENT ::= IDE.METH_IDE(PARAM) |
                  this.METH_IDE(PARAM) |
                  new CLASS_IDE(PARAM)

PARAM ::= EXP |
          ε

OP ::= + | - | * | / | == | != | < | <= | > | >= | & | && | ···

UNARYOP ::= !  | - | +

CONST_VAL ::= NUM | BOOL | null

BOOL ::= true | false
```

Le categorie sintattiche `NUM` e `IDE` rappresentano rispettivamente le stringhe numeriche e quelle alfanumeriche (che iniziano con almeno un carattere alfabetico). Le categorie sintattiche `CLASS_IDE` e `METH_IDE` coincidono con la categoria `IDE`. Si è preferito mantenerle separate in quanto a volte può essere comodo distinguere in modo esplicito identificatoti di classi, di metodo e di variabili. Comunque, siccome il nome del costruttore coincide con quello della classe, si è preferito utilizzare `CLASS_IDE` in `CONSTRUCTOR_DECL`. La categoria sintattica `IDE` non può però generare identificatore che contengono caratteri speciali come ad esempio <, >, &, !, ?,), etc. Quindi identificatori come `meth<`, `<init>`, `pluto!`, `pippo&pluto`, `(meth)` non sono identificatori validi.

Osservazione: Prendendo in esame la categoria sintattica `METH_BODY` si nota che i metodi che ritornano un valore devono avere il `return` statement alla fine del corpo del metodo. Inoltre, prendendo in esame la categoria sintattica `BASIC_STATEMENT`, si nota invece che `return` non ne fa parte. Questa semplificazione non permette quindi di scrivere lo statement seguente:

```
{
int n = 0;
...
if (n == 0)
    return 1;
else
    return n;
}
```

Comunque, lo statement precedente può essere riscritto nel modo seguente:

```
{
int n = 0;
int temp = 0;
...
if (n == 0)
    temp = 1;
else
    temp = n;
return temp;
}
```

Quindi, sebbene questa semplificazione limita la scrittura di codice in modo più conciso e leggibile, per lo scopo di queste note non rappresenta una limitazione in quanto è sempre possibile scrivere un metodo equivalente con il `return` alla fine del suo corpo.

Si noti, inoltre, la presenza della parola chiave `this`. Questa parola chiave identifica una variabile che, in ogni momento, contiene il riferimento all'oggetto

corrente e di conseguenza il suo tipo è il tipo dell'oggetto corrente (cioè la classe nella quale la parola `this` è usata). In +/-Java questa parola può essere utilizzata solo all'interno della dichiarazione di una classe, ed in particolare, nel corpo di un metodo oppure di un costruttore. La parola `this` deve essere utilizzata solo ed esclusivamente per riferirsi ad una variabile oppure ad un metodo della classe stessa. Il codice seguente mostra un esempio:

```
01:class Coord {
02:  private int x;
03:  private int y;
04:  private int sommaCoord = this.x + this.y; (NON AMMESSO IN +/-Java)
05:
06:  public void set_prima_coordinata(int l) {
07:    this.x = l;
08:  }
09:
10:  public void set_seconda_coordinata(int a) {
11:    this.y = a;
12:  }
13:
14:  public void set_prima_e_seconda_coordinata(int b) {
15:    this.set_prima_coordinata(b);
16:    this.set_seconda_coordinata(b);
17:  }
18:  ...
19:}
```

In Java questa parola chiave potrebbe essere utilizzata anche nella inizializzazione di una variabile ma, come mostrato nel codice di riga `04`, ciò non è ammesso in +/-Java. Infatti, tra le varie restrizioni, si ha che le variabili di istanza sono tutte `private` e non possono essere inizializzate contestualmente alla loro dichiarazione (vedi Sezione 4.1). Di conseguenza non è *a priori* ammesso avere dichiarazioni come quella di riga `04`[6].

Studiando con attenzione la grammatica precedente è possibile derivare osservazioni come le precedente, e capire le altre semplificazioni e restrizioni che +/-Java impone a livello sintattico al linguaggio Java come da specifica ufficiale.

[6]La possibilità di avere questo tipo di dichiarazioni richiederebbe una trattazione generale dei cosiddetti *inizializzatori di instanza* (*instance initializers*) e degli *inizializzatori di variabili instanza* (*instance variable initializers*), ma ciò esula gli scopi di queste note.

5 Struttura dello stato

In questa sezione richiameremo le nozioni ed estenderemo i formalismi presentati nelle note *Semantica Operazionale di R. Barbuti, P. Mancarella e F. Montangero* (da pag. 1 a pag. 39) oppure nella rivisitazione *Elementi di Semantica Operazionale - Appunti per gli studenti di Fondamenti di Programmazione - Corso di Laurea in Informatica, Universita' di Pisa - A.A. 2004/05 di R. Barbuti, P. Manacarella e F. Turini* (da pag. 1 a pag. 48) per poter trattare la programmazione orientata agli oggetti.

Avremo bisogno di una struttura dello stato più sofisticata dove le associazioni identificatori valori saranno **solo una parte** della struttura stato, che chiameremo *stato principale*. In particolare, per poter trattare correttamente caratteristiche più complesse del linguaggio, quali l'**annidamento dei blocchi** e la **chiamata a metodo**, è stato necessario dare allo stato principale una organizzazione più complessa. Intuitivamente, invece di trattarlo come una singola tabella piatta di associazioni tra identificatori e valori, lo tratteremo come una pila, ovvero come una sequenza di tabelle, ciascuna delle quali è un insieme di associazioni identificatori valori. Il termine *frame* è usato per far riferimento a una singola tabella.

In Figura 2, si consideri la rappresentazione grafica dello stato principale ρ costituito da due frame φ_1 e φ_2. L'identificatore x ha due valori nello stato principale, uno nel frame φ_1 ed uno nel frame φ_2. Qual è il valore di x nello stato principale ρ sopra indicato? È il primo valore che troviamo associato a x, ricercando x a partire dal frame più recente: il più vicino al top della pila. Ricordiamo che i frame vengono sempre aggiunti al ed eliminati dal top della pila. Nell'esempio precedente φ_2 è stato impilato su φ_1 e il valore di x risulta quindi 32. Da questo momento in poi, chiameremo lo stato principale *stack* in modo da richiamare la sua struttura a pila. Come vedremo a breve, la componente stack sarà quindi solo una parte della struttura completa dello stato. Passiamo ora alla definizione formale dello *stack*.

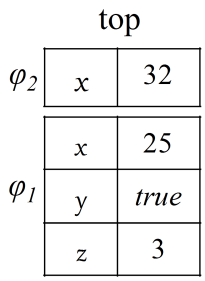

Figura 2: Stato principale ρ

Un frame φ è una funzione che associa ad un numero finito di identificatori un valore diverso da $\perp$, e $\perp$ a tutti gli altri:

$$\varphi : \texttt{IDE} \to Val \cup \{\bot\}$$
$$\text{dove } Val = \mathbb{Z} \cup \mathbb{B}$$

Come specificato nella Sezione 4, `IDE` è una categoria sintattica e non un insieme. Formalmente possiamo comunque dire che l'insieme di tutti gli identificatori sintatticamente corretti è generato dalla categoria sintattica `IDE`. Abusando di notazione scriveremo quindi $x \in \texttt{IDE}$ per indicare che x è un identificatore che appartiene all'insieme di tutti gli identificatori sintatticamente corretti.

Come vedremo di seguito, per permettere la trattazione delle classi e degli oggetti, l'insieme Val sarà arricchito in modo da includere non solo tipi primitivi ma anche tipi di riferimento (introdotti in Sezione 3).

Uno stack ρ è quindi una sequenza di frame. Denotando con Ω lo *stack vuoto* e indicando con Φ l'insieme dei frame ammissibili, l'insieme $\mathcal{S}$ degli *stack* legali è definito come:

$$\mathcal{S} = \{\Omega\} \cup \{\varphi \cdot \rho \mid \varphi \in \Phi,\ \rho \in \mathcal{S}\}$$

Le operazioni per inserire, eliminare e recuperare il frame al top dello stack in seguito si chiameranno rispettivamente *push* e *pop*. Esse sono così definite:

$$\begin{aligned} push &: \Phi \times \mathcal{S} \to \mathcal{S} \\ pop &: \mathcal{S} \to \mathcal{S} \end{aligned}$$

Dato un frame $\varphi \in \Phi$ e uno stack $\rho \in \mathcal{S}$, $push(\varphi, \rho) = \varphi{\cdot}\rho$. Per esempio, per lo stack in Figura 2 si ha che $push(\varphi_2, push(\varphi_1, \Omega)) = \varphi_2{\cdot}\varphi_1{\cdot}\Omega$.

Dato uno stack $\rho' = \varphi{\cdot}\rho$, $pop(\rho') = \rho$. Per esempio, per lo stack in Figura 2 si ha che $\rho' = \varphi_2{\cdot}\varphi_1{\cdot}\Omega$ e $pop(\rho') = \varphi_1{\cdot}\Omega$.

Con abuso di notazione[7], l'*accesso* ad un identificatore nello stack ρ è definito come:

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

Definiamo ora la semantica della *modifica* del valore associato ad un identificatore nello stack ρ.

$$\rho[{}^{v}/_{x}] = \begin{cases} \Omega & \text{se } \rho = \Omega \\ \varphi[{}^{v}/_{x}] \cdot \rho' & \text{se } \rho = \varphi \cdot \rho' \text{ e } \varphi(x) \neq \bot \\ \varphi \cdot \rho'[{}^{v}/_{x}] & \text{se } \rho = \varphi \cdot \rho' \text{ e } \varphi(x) = \bot \end{cases}$$

[7] ρ è una sequenza e non una funzione!

Osserviamo che le scritture $\rho[^v/_x]$ e $\varphi[^v/_x]$ determinano due comportamenti differenti sullo stack: la seconda modifica sempre lo stack in quanto modifica il frame più recente dello stack, mentre la prima può modificare un frame meno recente o addirittura potrebbe non avere effetto sullo stack se la variabile di cui si vuole modificare il valore associato non è stata mai dichiarata. Infine ricordiamo che ω è il frame vuoto che associa $\perp$ a tutti gli identificatori:

$$\omega : \texttt{IDE} \to \{\perp\} \quad \text{dove } \omega \in \Phi.$$

5.1 Stato e oggetti

Il modello di stato considerato finora consente di trattare variabili con tipo basico: interi e booleani. In Java ogni tipo complesso, a partire dalle stringhe, è trattato con il paradigma degli oggetti. Come già detto, la natura di questi oggetti è definita dalle classi che ne dichiarano la struttura come insieme di variabili e di metodi che ne permettono la modifica. Può essere utile ricordare che in Java si usa spesso il termine *campo* per indicare una variabile di una classe e il termine *membro* per indicare sia una variabile sia un metodo di una classe.

Per trattare le classi e gli oggetti è necessario arricchire lo stato con altre due componenti. In questa sezione faremo una breve introduzione dei nuovi concetti di cui avremo bisogno per trattare le classi e gli oggetti. Questi concetti verranno poi approfonditi nelle sezioni che seguono.

Da questo punto in poi lo stato σ verrà visto come la terna $< \rho, \mu, \delta >$, le cui componenti ρ, μ e δ rappresentano rispettivamente lo *stack*, l'*heap* e la *library*.

$$\rho : \texttt{IDE} \to Val^* \cup \{\perp\} \qquad dove \;\; Val^* = Val \cup Loc \cup \{\texttt{null}\}$$

$$\mu : Loc \to (\Phi \times \Phi_m) \cup \{\perp\}$$

$$\delta : \texttt{CLASS\_IDE} \to (\Phi \times \Phi_m) \cup \{\perp\}$$

La *library* δ associa al nome delle classi una loro rappresentazione e viene costruita una volta per tutte utilizzando le dichiarazioni di tutte le classi, prima dell'esecuzione del programma vero e proprio. Più in dettaglio, la dichiarazione di una classe di nome `C` ha l'effetto di aggiungere in δ un'associazione tra l'identificatore della classe (cioè `C`) ed una coppia di frame $< \varphi_{v(C)}, \varphi_{m(C)} > \in \Phi \times \Phi_m$ che rappresentano rispettivamente le variabili (o campi) ed i metodi definiti all'interno della classe `C` stessa. Si noti l'introduzione del nuovo simbolo Φ_m che rappresenta l'insieme di tutti i metodi ammissibili e verrà definito formalmente nella Sezione 6.

Il comportamento dello *stack* ρ è quello definito nella sezione precedente, con un nuovo accorgimento legato alla trattazione delle istanze di classe (cioè gli oggetti). In particolare, l'insieme Val^* arricchisce l'insieme Val per includere anche l'insieme

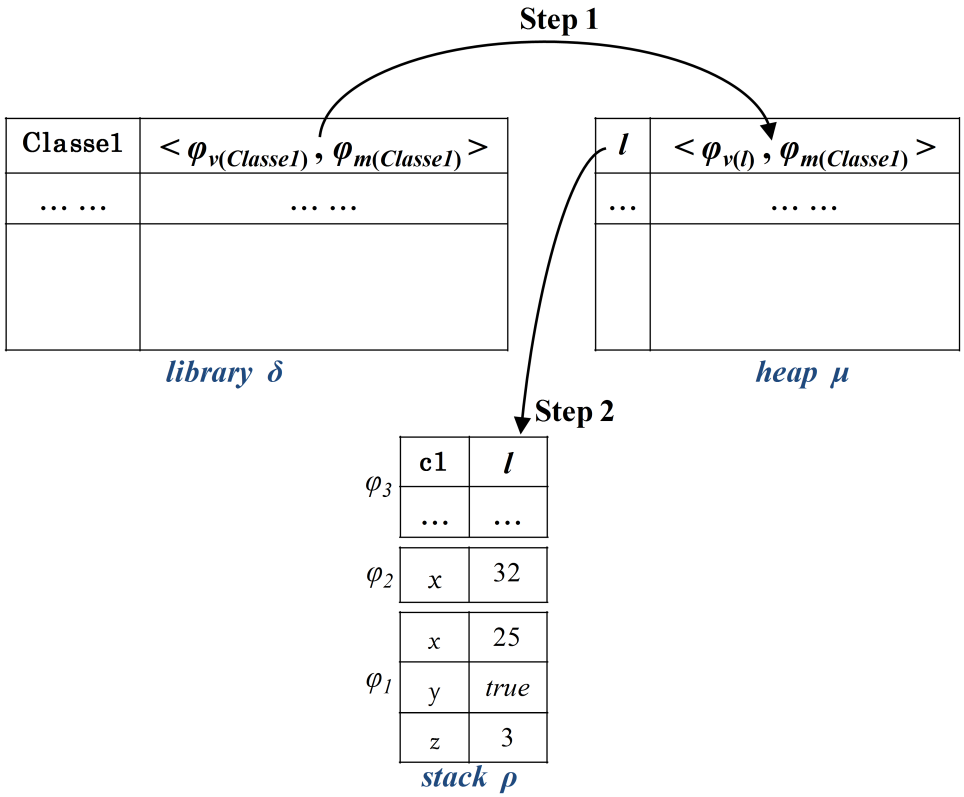

Figura 3: Effetto di "`Classe1 c1 = new Classe1()`"

Loc dei valori di riferimento e l'elemento `null`. Infatti, la JVM manipola due tipi di valori che possono essere assegnati a variabili, passati come argomenti a metodi, ritornati da metodi, etc. Questi valori sono *valori primitivi* (come valori interi e booleani) e *valori riferimento* (indirizzi di memoria - o locazioni - per riferire oggetti nell'heap).

Informalmente, in Figura 3, osserviamo cosa accade quando si istanzia un oggetto mediante l'istruzione "`Classe1 c1 = new Classe1()`". Si può osservare che, al momento della dichiarazione della variabile riferimento `c1` (cioè una variabile di *tipo riferimento alla classe* `Classe1`), viene aggiunta una coppia nel frame più recente φ_3 dello stack corrente ρ. La coppia associa l'identificatore `c1` e l. Il valore l è il riferimento all'oggetto e cioè l'indirizzo di una locazione di memoria. Come sarà presto chiaro al lettore, tale riferimento rappresenta un puntatore alla struttura effettiva dell'oggetto stesso che viene materialmente memorizzata in μ. Nello specifico, si aggiunge in μ un'associazione tra la locazione l e una coppia di frame $< \varphi_{v(l)}, \varphi_{m(Classe1)} > \in \Phi \times \Phi_m$ che, copiati dalla library δ, rappresentano rispettivamente le variabili e i metodi della specifica istanza della classe `C` riferita da l (e quindi da `c1`). Se la dichiarazione di una variabile riferimento non prevede

l'inizializzazione, e quindi se si ha solo "`Classe1 c1`", il riferimento associato è il riferimento speciale *null*.

Ricapitolando, l'esecuzione della precedente `new` ha i seguenti effetti:

Step 1 Creazione in μ dello spazio di memoria riferito dalla locazione l per la struttura "personale" dell'oggetto mediante una coppia di frame $\langle \varphi_{v(l)}, \varphi_{m(Classe1)} \rangle$ contenente una copia dei frame $\varphi_{v(Classe1)}$ e $\varphi_{m(Classe1)}$ associati all'identificatore `Classe1` nella library δ. É utile osservare che copiando il frame $\varphi_{v(Classe1)}$ si è provveduto a cambiare il pedice in "$_{v(l)}$" ad indicare che in $\varphi_{v(l)}$ ci sono le variabili personali dell'oggetto alla locazione l. Questo cambiamento non serve per il frame $\varphi_{m(Classe1)}$ in quanto tutti gli oggetti della `Classe1` utilizzano sempre la stessa definizione dei metodi della `Classe1`.

Step 2 Inserimento di una nuova associazione nel frame più recente φ_3 dello *stack* per l'identificatore di variabile `c1` e la locazione l del passo precedente. Graficamente quello che avviene con i passi **1** e **2** è rappresentato in Figura 3.

Step 3 Chiamata al costruttore della classe. Si ricorda che il costruttore è un metodo chiamato subito dopo la creazione di un oggetto. Si ricorda inoltre che il metodo costruttore ha lo stesso nome della classe che lo dichiara. In particolare, la sua dichiarazione è molto simile alla dichiarazione di un metodo "standard" ma NON deve avere specificato il valore di ritorno. Se una classe non dichiara almeno un costruttore allora per default ne viene fornito uno che non prende argomenti.

Osservazione: Come è evidente dalla discussione precedente, la considerazione di variabili di tipo riferimento e di locazioni implica la ridefinizione dell'insieme Φ dei frame ammissibili. Infatti, riferendo di nuovo Figura 3, il frame φ_3 nello stack (come pure φ_1 e φ_2), il frame $\varphi_{v(Classe1)}$ nella library e il frame $\varphi_{v(l)}$ nell'heap possono ora associare identificatori di variabili non solo a interi e booleani ma anche a locazioni. Comunque, come formalizzato in Sezione 6, i frame φ_3, $\varphi_{v(Classe1)}$ e $\varphi_{v(l)}$ sono definiti allo stesso modo. Infatti, tutti associano identificatori di variabili a valori in Val^* e tutti quindi "hanno diritto" di appartenere all'insieme Φ.

- Per ogni classe `C`, il pedice "$_{v(C)}$" del relativo frame $\varphi_{v(C)}$ in δ si utilizza per indicare che il frame contiene solo le variabili definite come campi di `C`. Poi, come già detto, per ogni oggetto della classe `C`, nel momento della sua istanziazione, si crea in μ il frame $\varphi_{v(l)}$ delle sue variabili personali copiando il frame $\varphi_{v(C)}$ da δ. Di conseguenza,

- per ogni oggetto, il pedice "$_{v(l)}$" del relativo frame $\varphi_{v(l)}$ in μ si utilizza per indicare che il frame contiene solo le variabili personali dell'oggetto alla locazione l. Ogni oggetto ha quindi un proprio frame $\varphi_{v(l)}$ in μ.

- I frame φ_1, φ_2, φ_3, ... nello stack ρ si utilizzano per le variabili dichiarate localmente all'interno di qualsiasi metodo oppure blocco.

Differentemente dalle variabili, per ogni classe `C` in δ il frame dei suoi metodi $\varphi_{m(C)}$ in δ e le sue copie per tutte le sue istanze in μ appartengono al nuovo insieme Φ_m. Infatti, questi frame non possono essere inclusi in Φ in quanto associano identificatori di metodi non a semplici valori in Val^* ma ad una struttura più complessa che verrà definita in Sezione 6.

6 Semantica operazionale di +/-Java

Questa sezione richiama e aggiunge alcune definizioni che verranno utilizzate in seguito per la definizione della nostra semantica operazionale dinamica e quindi per la definizione di un sistema di transizione che, a fronte di un programma Java (e quindi del suo bytecode), passa di configurazione in configurazione mostrando un comportamento operativo consistente con quello della JVM (o meglio una sua semplificazione).

Si ricorda che un sistema di transizioni è una tripla $\langle\, \Gamma,\, T,\, \rightarrow \,\rangle$ dove:

- Γ è un insieme i cui elementi sono detti *configurazioni*;
- $T \subseteq \Gamma$ è un insieme di configurazioni dette *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.

L'insieme Φ, come esteso precedentemente per trattare gli oggetti, ed in particolare per distinguere (***i***) le variabili definite localmente all'interno di metodi oppure blocchi, (***ii***) le variabili definite come campi di una classe `C` e (***iii***) le variabili delle diverse istanze di `C`, è l'insieme di tutte le funzioni che modellano associazioni del tipo:

(i) $\varphi : \texttt{IDE} \rightarrow Val^* \cup \{\bot\}$
dove $Val^* = Val \cup Loc \cup \{\texttt{null}\}$

(ii) $\varphi_{v(C)} : \texttt{IDE} \rightarrow Val^* \cup \{\bot\}$
dove $Val^* = Val \cup Loc \cup \{\texttt{null}\}$ e $\texttt{C} \in \texttt{CLASS\_IDE}$

(iii) $\varphi_{v(l)} : \texttt{IDE} \rightarrow Val^* \cup \{\bot\}$
dove $Val^* = Val \cup Loc \cup \{\texttt{null}\}$ e $l \in Loc$

(iv) $\omega : \texttt{IDE} \rightarrow \{\bot\}$

Come già anticipato, φ, $\varphi_{v(C)}$ e $\varphi_{v(l)}$ identificano praticamente la stessa tipologia di frame ma si è preferito usare pedici parametrici differenti per separare logicamente i frame all'interno dello stack dai frame all'interno della library e dell'heap. Questa intuitiva distinzione diventerà ancora più esplicita in seguito studiando il sistema di transizione per +/- Java.

Concretamente, per una data classe `C` (oppure interfaccia[8]), il frame dei campi $\varphi_{v(C)}$ in δ e le tutte sue copie $\varphi_{v(l_1)}, \ldots, \varphi_{v(l_n)}$ per le diverse istanze di `C` in μ modellano (solo in parte) la struttura di una tabella che in Java (e precisamente nella JVM) è chiamata *fields table* (tabella dei campi). In realtà, per ogni oggetto della classe `C` e per ogni suo campo, questa tabella contiene un insieme di informazioni chiamate *field_info* ma, per il nostro +/-Java, ogni frame $\varphi_{v(l_i)}$ prende in input un identificatore di campo e ritorna semplicemente il suo valore in Val^* (oppure $\bot$ se il campo non è stato precedentemente dichiarato). Per esempio, considerando la classe `Persona` all'inizio di queste note e supponendo che la sua istanza per Mario Rossi si trovi alla locazione l di μ, si ha che $\varphi_{v(l)}(\texttt{nome}) =$ Mario e $\varphi_{v(l)}(\texttt{cognome}) =$ Rossi.

L'insieme Φ_m, come introdotto precedentemente per la trattazione degli oggetti ed in particolare per la trattazione dei metodi definiti all'interno di una classe `C`, è l'insieme di tutte le funzioni che modellano associazioni del tipo:

- $\varphi_{m(C)} : \texttt{METH\_IDE} \cup \{\texttt{<init>}\} \rightarrow Method \cup \{\bot\}$

  dove $Method = \left\{ < R, I, M > \mid \begin{array}{l} R \in \texttt{TYPE\_NAME} \cup \{\texttt{void}\}, \\ I \in \texttt{IDE} \cup \{\bot\}, \\ M \in \texttt{METH\_BODY} \cup \\ \qquad \texttt{METH\_BODY\_NO\_RETURN} \end{array} \right\}$

  Concretizzando di nuovo verso il mondo Java, per una data classe `C`, la funzione $\varphi_{m(C)}$ modella (in parte) la struttura di una tabella associata a `C` che in Java è chiamata *methods table* (tabella dei metodi). La funzione $\varphi_{m(C)}$ prende in input un identificatore di metodo e ritorna la sua descrizione. Come spiegato più avanti, il nome speciale `<init>` è fornito di default dal compilatore Java ed è utilizzato dalla JVM per identificare il metodo costruttore.

  Ogni tripla $<$ `T, x, MB` $> \in Method$ rappresenta la descrizione completa della struttura del metodo e modella (parte di) una struttura che in Java ufficiale è chiamata *method_info*. Quindi gli elementi *method_info* fanno parte della *methods table*. Nello specifico, `T` $\in R$ è il tipo di ritorno del

[8] Il nostro +/-Java non tratta le interfaccie.

metodo oppure `void` qualora il metodo non ha valore di ritorno; `x` $\in I$ è il parametro formale in ingresso[9]; `MB` $\in M$ è il corpo del metodo contenente tutte le istruzioni che definiscono il metodo.

Per esempio, considerando di nuovo la classe `Persona`, si ha che:
• $\varphi_{m(Persona)}$ (`get_indirizzo`) =< `String`, $\perp$, `return this.indirizzo;` >
• $\varphi_{m(Persona)}$ (`set_indirizzo`) =< `void`, `indirizzo`, `this.indirizzo = indirizzo;` >.

In particolare, data una classe `C` e l'identificatore `meth` di un suo metodo, se $\varphi_{m(C)}$(`meth`) = $\perp$ significa che `meth` non è stato definito nella classe `C` in considerazione. Allo stesso modo, dato l'identificatore speciale `<init>`, se $\varphi_{m(C)}$(`<init>`) = $\perp$ significa che il costruttore non è stato esplicitamente definito per la classe `C` e ne verrà fornito uno di default.

Allo stesso modo dell'insieme Φ, $\omega \in \Phi_m$ è il frame vuoto che associa $\perp$ a tutti gli identificatori di metodo:

$$\omega : \texttt{METH\_IDE} \rightarrow \{\perp\}$$

L'insieme degli *stack* legali $\mathcal{S}$, come definito precedentemente in Sezione 5 ed esteso in Sezione 5.1, comprende tutte le funzioni del tipo

- $\rho : \texttt{IDE} \rightarrow Val^* \cup \{\perp\}$ dove $Val^* = Val \cup Loc \cup \{\texttt{null}\}$

L'insieme degli *heap* legali $\mathcal{M}$ comprende tutte le funzioni del tipo

- $\mu : Loc \rightarrow (\Phi \times \Phi_m) \cup \{\perp\}$

  La funzione μ prende in input una locazione di memoria l e ritorna la coppia di frame $\mu(l) = \langle \varphi_{v(l)}, \varphi_{m(C)} \rangle$ se in μ esiste un'associazione tra l e un'istanza della classe `C`, $\perp$ altrimenti. Si ricorda che il pedice "$_{v(l)}$" lega lo specifico oggetto nella locazione l alle sue variabili "personali", mentre il pedice "$_{m(C)}$" lega qualunque oggetto della classe `C` ai medesimi metodi definiti in `C`.

L'insieme delle *library* legali Δ comprende tutte le funzioni del tipo

- $\delta : \texttt{CLASS\_IDE} \rightarrow (\Phi \times \Phi_m) \cup \{\perp\}$

  La funzione δ prende in input un identificatore di classe `C` e ritorna la rappresentazione $\delta(\texttt{C}) = \langle \varphi_{v(C)}, \varphi_{m(C)} \rangle$ se la classe `C` è stata precedentemente dichiarata, $\perp$ altrimenti. Sebbene in Java è possibile definire più classi identificate dallo stesso nome (ma in packages differenti!), nel nostro +/-Java ciò non è permesso.

[9] Si considerano metodi con un solo parametro formale di ingresso.

Quindi, l'insieme degli stati ammissibili è così definito

$$\texttt{State} = \{\sigma \mid \sigma \in \mathcal{S} \times \mathcal{M} \times \Delta\}.$$

Questa sezione si conclude definendo alcune funzioni che risulteranno molto utili nella presentazione del sistema di transizione per + Java.

- La funzione *defval* che prende un tipo di dato e restituisce il relativo valore di default è così definita:

$$\mathit{defval}(\texttt{T}) = \begin{cases} \underline{0} & \text{se} \quad \texttt{T} = \texttt{int} \\ \underline{\mathit{false}} & \text{se} \quad \texttt{T} = \texttt{boolean} \\ \underline{\mathit{null}} & \text{se} \quad \texttt{T identifica una Classe} \\ \bot & \text{altrimenti} \end{cases}$$

- La funzione di valutazione semantica η che prende una rappresentazione sintattica di un valore costante e restituisce il corrispondente valore semantico è così definita:

$$\eta : \texttt{CONST\_VAL} \to \mathbb{Z} \cup \mathbb{B} \cup \{\underline{\mathit{null}}\}$$

$$\begin{gathered} \eta(\texttt{0}) = \underline{0} \\ \eta(\texttt{1}) = \underline{1} \\ \eta(\texttt{2}) = \underline{2} \\ \cdots \\ \eta(\texttt{-0}) = \underline{-0} \\ \eta(\texttt{-1}) = \underline{-1} \\ \eta(\texttt{-2}) = \underline{-2} \\ \cdots \\ \eta(\texttt{true}) = \underline{\mathit{true}} \\ \eta(\texttt{false}) = \underline{\mathit{false}} \\ \eta(\texttt{null}) = \underline{\mathit{null}} \end{gathered}$$

- La funzione $\mathit{copy}(\langle \varphi_{v(C)}, \varphi_{m(C)} \rangle, l)$ prende in input la rappresentazione $\delta(\texttt{C}) = \langle \varphi_{v(C)}, \varphi_{m(C)} \rangle$ di una classe C, una locazione di memoria l e ritorna in $\langle \varphi_{v(l)}, \varphi_{m(C)} \rangle$ la copia di $\langle \varphi_{v(C)}, \varphi_{m(C)} \rangle$ dopo aver ridenominato il pedice "$_{v(C)}$" in "$_{v(l)}$".

- La funzione $\mathit{newloc}(\mu, \texttt{C})$ prende in input l'heap μ e il nome di una classe C e ritorna l'indirizzo di una locazione l per l'area di memoria in μ capace di memorizzare la coppia $\langle \varphi_{v(l)}, \varphi_{m(C)} \rangle$ di un'istanza di C.

6.1 Il sistema *prog*

$$\begin{array}{rcl} \Gamma_{prog} & = & \{\langle\, \mathtt{P},\, \sigma \,\rangle \mid \mathtt{P} \in \mathtt{PROGRAM},\ \sigma \in \mathtt{State}\} \cup \\ & & \{\sigma \mid \sigma \in \mathtt{State}\} \\ T_{prog} & = & \{\sigma \mid \sigma \in \mathtt{State}\} \end{array}$$

Il sistema *prog* utilizza lo stato $\sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle$ che rappresenta lo stato iniziale vuoto, così composto:

$\rho_0 = \Omega$ dove Ω è lo stack vuoto.
$\mu_0 : Loc \to \{\bot\}$ è l'heap vuoto che associa $\bot$ a tutte le locazioni della memoria.
$\delta_0 : \mathtt{CLASS\_IDE} \to \{\bot\}$ è la library vuota che associa $\bot$ a tutti gli identificatori di classe.

Si ricorda che $\omega \in \Phi$ è il frame vuoto così definito $\omega : \mathtt{IDE} \to \{\bot\}$.

$$\frac{\begin{array}{c} \sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle \\ \langle\, \mathtt{C}\,, \sigma_0 \,\rangle \longrightarrow_{decl} \sigma_1 \quad \sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle \quad \sigma_2 = \langle\, \omega[\,\overline{null}/_{\mathtt{I}}\,] \cdot \Omega,\, \mu_0,\, \delta_1 \,\rangle \\ \langle\, \mathtt{D}\,, \sigma_2 \,\rangle \longrightarrow_{local\_decl} \sigma_3 \quad \langle\, \mathtt{S}\,, \sigma_3 \,\rangle \longrightarrow_{com} \sigma' \end{array}}{\begin{array}{l} \mathtt{C\ public\ class\ Program\ \{} \\ \langle\ \ \mathtt{public\ static\ void\ main(String[]\ I)} \\ \ \ \mathtt{\{D\ S\}} \\ \mathtt{\}} \end{array} \ , \sigma_0 \,\rangle \longrightarrow_{prog} \sigma'} \quad (prog)$$

A titolo di esempio, nel caso in cui non vi siano dichiarazioni `D` all'interno del corpo del metodo `main`, la regola *prog* diventa:

$$\frac{\begin{array}{c} \sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle \\ \langle\, \mathtt{C}\,, \sigma_0 \,\rangle \longrightarrow_{decl} \sigma_1 \quad \sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle \quad \sigma_2 = \langle\, \omega[\,\overline{null}/_{\mathtt{I}}\,] \cdot \Omega,\, \mu_0,\, \delta_1 \,\rangle \\ \langle\, \mathtt{S}\,, \sigma_2 \,\rangle \longrightarrow_{com} \sigma' \end{array}}{\begin{array}{l} \mathtt{C\ public\ class\ Program\ \{} \\ \langle\ \ \mathtt{public\ static\ void\ main(String[]\ I)} \\ \ \ \mathtt{\{S\}} \\ \mathtt{\}} \end{array} \ , \sigma_0 \,\rangle \longrightarrow_{prog} \sigma'} \quad (prog)$$

Nelle due regole precedenti, lo stack $\omega[\,\overline{null}/_{\mathtt{I}}\,] \cdot \Omega$ deriva da uno stack vuoto Ω su cui è stato "impilato" un frame vuoto ω modificato con l'associazione $\overline{null}/_{\mathtt{I}}$.

Come introdotto in Sezione 2.2.1, una applicazione di nome *java* prende in input un programma scritto in Java (o meglio il suo bytecode nei file *.class*) e lancia una istanza della JVM eseguendo il programma su di essa. La JVM inizia la sua

esecuzione invocando il metodo `main` della classe principale specificata (`Programm` nel nostro caso) e passando ad esso un singolo argomento (`I` nel nostro caso) che è un array di stringhe. Queste stringhe possono poi essere utilizzate e manipolate dal programma. Quindi, associare `I` sempre a *null* evidenzia un'altra restrizione imposta al nostro +/-Java che non considera il passaggio di parametri al metodo *main*.

6.2 Il sistema *decl*

$$\begin{aligned}\Gamma_{decl} &= \{\langle\, \texttt{CD},\, \sigma\,\rangle \mid \texttt{CD} \in \texttt{CLASS\_DECL\_LIST},\ \sigma \in \texttt{State}\} \cup \\ &\quad \{\sigma \mid \sigma \in \texttt{State}\} \\ T_{decl} &= \{\sigma \mid \sigma \in \texttt{State}\}\end{aligned}$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta\,\rangle \\ \langle\, \texttt{ML}\,, \langle\, \omega,\, \omega\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \\ \delta' = \delta[\,{}^{\langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle}/_{\texttt{C}}\,] \quad \sigma' = \langle\, \rho,\, \mu,\, \delta'\,\rangle\end{array}}{\langle\, \texttt{class C}\,\{\texttt{ML}\}\,, \sigma\,\rangle \longrightarrow_{decl} \sigma'} \quad (decl_{class})$$

$$\frac{\texttt{C} \in \texttt{CLASS\_DECL} \quad \langle\, \texttt{C}\,, \sigma\,\rangle \longrightarrow_{decl} \sigma_1 \quad \langle\, \texttt{CLIST}\,, \sigma_1\,\rangle \longrightarrow_{decl} \sigma'}{\langle\, \texttt{C CLIST}\,, \sigma\,\rangle \longrightarrow_{decl} \sigma'} \quad (decl_{concat})$$

I sistema ***decl*** formalizza la dichiarazione di una o più classi come listate prima della classe `Program`. La regola $decl_{class}$, formalizza la dichiarazione di una singola classe. Considerando la lista di tutti i membri `ML` di una classe identificata da `C`, si aggiunge un'associazione nella library δ tra l'identificatore `C` e la coppia $\langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle$ della *fields table* e della *methods table*, rispettivamente. Tale coppia si ricava con il sistema ***cd*** applicato alla lista dei membri `ML` partendo da due tabelle vuote $\langle\, \omega,\, \omega\,\rangle$.

Nella regola $decl_{concat}$, `C CLIST` rappresenta una lista di classi avente `C` come prima classe della lista. La regola $decl_{concat}$ utilizza dapprima la regola $decl_{class}$ per dichiarare la classe `C` e poi richiama ricorsivamente se stessa per dichiarare tutte le restanti classi in `CLIST`.

6.3 Il sistema *cd*

$$\begin{aligned}
\Gamma_{cd} &= \{\langle\, \texttt{ML},\, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \mid \texttt{ML} \in \texttt{CLASS_MEMBER_DECL_LIST},\\
&\qquad\qquad \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \in \Phi \times \Phi_m,\ C \in \texttt{CLASS_IDE}\} \cup\\
&\quad \{\langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \mid \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \in \Phi \times \Phi_m,\ C \in \texttt{CLASS_IDE}\}\\
T_{cd} &= \{\langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \mid \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle \in \Phi \times \Phi_m,\ C \in \texttt{CLASS_IDE}\}
\end{aligned}$$

$$\frac{\underline{v} = \mathit{defval}(\texttt{T}) \quad \varphi'_{v(C)} = \varphi_{v(C)}[\,{}^{\underline{v}}/_{\texttt{x}}\,]}{\langle\, \texttt{private T x;}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi'_{v(C)},\, \varphi_{m(C)}\,\rangle} \quad (cd_{var})$$

$$\frac{\varphi'_{m(C)} = \varphi_{m(C)}[\,{}^{\langle\, \texttt{T}_\texttt{r},\, \bot,\, \texttt{MB}\,\rangle}/_{\texttt{m}}\,]}{\langle\, \texttt{public T}_\texttt{r}\ \texttt{m() MB}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi'_{m(C)}\,\rangle} \quad (cd_{meth\_nopar})$$

$$\frac{\varphi'_{m(C)} = \varphi_{m(C)}[\,{}^{\langle\, \texttt{T}_\texttt{r},\, \texttt{x},\, \texttt{MB}\,\rangle}/_{\texttt{m}}\,]}{\langle\, \texttt{public T}_\texttt{r}\ \texttt{m(T}_\texttt{p}\ \texttt{x) MB}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi'_{m(C)}\,\rangle} \quad (cd_{meth})$$

$$\frac{\varphi'_{m(C)} = \varphi_{m(C)}[\,{}^{\langle\, \texttt{void},\, \bot,\, \texttt{MB}\,\rangle}/_{\texttt{<init>}}\,]}{\langle\, \texttt{public C() MB}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi'_{m(C)}\,\rangle} \quad (cd_{constr\_nopar})$$

$$\frac{\varphi'_{m(C)} = \varphi_{m(C)}[\,{}^{\langle\, \texttt{void},\, \texttt{x},\, \texttt{MB}\,\rangle}/_{\texttt{<init>}}\,]}{\langle\, \texttt{public C(T}_\texttt{p}\ \texttt{x) MB}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi'_{m(C)}\,\rangle} \quad (cd_{constr})$$

$$\frac{\begin{array}{c}\texttt{M} \in \texttt{CLASS\_MEMBER\_DECL} \quad \langle\, \texttt{M}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi'_{v(C)},\, \varphi'_{m(C)}\,\rangle\\ \langle\, \texttt{MLIST}\,, \langle\, \varphi'_{v(C)},\, \varphi'_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi''_{v(C)},\, \varphi''_{m(C)}\,\rangle\end{array}}{\langle\, \texttt{M MLIST}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle\,\rangle \longrightarrow_{cd} \langle\, \varphi''_{v(C)},\, \varphi''_{m(C)}\,\rangle} \quad (cd_{concat})$$

Il sistema ***cd*** formalizza la dichiarazione dei membri in una classe `C` e quindi delle sue variabili, dei suoi metodi e del costruttore. In tale sistema le configurazioni terminali sono tutte della forma $\langle\, \varphi_{v(C)},\, \varphi_{m(C)}\,\rangle$ ad indicare che dichiarare membri significa aggiungere associazioni per essi nella *fields table* $\varphi_{v(C)}$ e nella *methods table* $\varphi_{m(C)}$ della classe in considerazione.

Nelle regole del sistema ***cd*** vale la pena notare che $\texttt{T}, \texttt{T}_\texttt{p} \in \texttt{TYPE\_NAME}$ e $\texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \cup \{\texttt{void}\}$ (si veda la grammatica riportata in Sezione 4.2). Infatti, nella specifica ufficiale del linguaggio Java si mantiene distinto `void` dagli altri tipi (ristretti ai tipi "in" `TYPE_NAME` per il nostro +/-Java). Quindi per Java `void` non è un tipo vero e proprio ma è utilizzato per indicare che un metodo non ha un valore di ritorno. Vi sono però linguaggi di programmazione come il C ed il C++ che danno la possibiltà di usare `void` come un vero e proprio tipo ma la discussione di ciò esula dagli obiettivi di queste note. Di seguito, in assenza di ambiguità, useremo la terminologia *tipo di ritorno* per indicare sia i tipi in `TYPE_NAME` che `void`.

Dichiarare una variabile `x`, come campo di una classe, con la regola cd_{var} significa creare una associazione per essa in $\varphi_{v(C)}$. La variabile `x` è associata al valore di default dipendentemente dal suo tipo `T`. In Java è anche possibile inizializzare una variabile, come campo di una classe, con il valore desiderato facendo un assegnamento esplicito contestuale alla dichiarazione. Si possono quindi scrivere dichiarazioni del tipo `private T x = E;`. Come già detto in Sezione 4, il nostro +/-Java non offre questa possibilità in quanto la gestione di questo tipo di inizializzazioni complicherebbe notevolmente la trattazione[10]. Come vedremo, con la nostra semantica è però possibile gestire inizializzazioni di variabili contestualmente alla loro dichiarazione all'interno di metodi e blocchi.

Dichiarare un metodo `m` con le regole cd_{meth_nopar} e cd_{meth} significa creare una associazione in $\varphi_{m(C)}$ tra l'identificatore di metodo `m` e una terna contenente il tipo di ritorno T_r, il parametro formale in ingresso `x` (oppure $\perp$ in sua assenza) e il corpo del metodo `MB`.

Per quanto riguarda la dichiarazione di un costruttore, sebbene molto simile a quella di metodo, bisogna fare alcune considerazioni. Nel linguaggio Java ufficiale, il costruttore è un metodo che non ha valore di ritorno, NON deve però avere specificato `void`. Inoltre, la gestione della dichiarazione del costruttore si mantiene separata dagli altri metodi generici. Per questo motivo, anche il nostro +/-Java gestisce separatamente la dichiarazione del costruttore per mezzo delle le regole cd_{constr_nopar} e cd_{constr}. In queste regole infatti si sostituisce l'identificatore `C` del costruttore (che è uguale al nome della classe) con un identificatore speciale `<init>` fornito di default dal compilatore Java. Come già detto nella Sezione 4.2, `<init>` non è un identificatore valido in quanto non può essere generato dalla categoria sintattica `IDE`. Come tale, `<init>` non può essere utilizzato come identificatore durante la scrittura di un programma in linguaggio Java. In considerazione di ciò, si può utilizzare `<init>` come identificatore "speciale" dedicato ai costruttori senza generare alcun problema.

È anche interessante notare che nelle regole cd_{constr_nopar} e cd_{constr}, e in particolare nell'associazione $\langle \texttt{void}, \perp, \texttt{MB} \rangle /_{\texttt{<init>}}$ di $\varphi_{m(C)}$, il primo elemento della tripla riporta `void` come tipo di ritorno del costruttore. Questo indica che, relativamente al valore di ritorno, la JVM tratta il costruttore come un metodo "standard" che non ha valore di ritorno.

Se si dichiarasse un metodo con lo stesso nome della classe (e quindi del costruttore) specificando il tipo del valore di ritorno (oppure `void`), il metodo verrebbe considerato come un metodo "standard" e per esso bisognerebbe applicare la regola cd_{meth_nopar} oppure la regola cd_{meth} e non le regole cd_{constr_nopar} e cd_{constr}. Quindi il nome del metodo non verrebbe sostituito con `<init>`. Sebbene questa situazione non rappresenta un errore, la maggior parte dei compilatori Java es-

[10]Per esempio, se consideriamo dichiarazioni di variabili d'istanza (e non dichiarazioni di variabili di classe del tipo `static T x = E`), dopo la traduzione del codice sorgente in bytecode, la loro inizializzazione viene effettuata direttamente nel costruttore subito prima di eseguire il suo corpo. Questo richiederebbe una gestione più complessa.

istenti avvertono il programmatore con un messaggio di *warning* (per esempio: *"Il metodo ha lo stesso nome del costruttore!"*) ad indicare che si tratta di una cattiva programmazione che può poi facilitare l'introduzione di errori veri e propri.

Dopo lo studio delle regole exp_{new_nopar}, exp_{new_par} e $exp_{new\_default}$ del sistema *exp*, sarà chiaro che per ogni classe `C` avere nella library δ (e più precisamente nella *methods table* $\varphi_{m(C)}$) il costruttore identificato con `<init>` permette alla nostra macchina formale astratta (ma anche alla JVM) di riconoscere il suo corpo `MB`, distinguerlo dagli altri metodi e trattare la sua chiamata (in luogo di una `new`) in modo speciale.

Sebbene non necessaria per il proseguimento della lettura, si è preferito riportare la seguente osservazione che al lettore interessato potrebbe tornare utile.

Osservazione: L'informazione sul ritorno dei metodi e dei costruttori è utilizzata a tempo di esecuzione dalla JVM per risolvere problemi legati alla chiamata dei metodi, per esempio, in caso di *ereditarietà* e *overriding*. Come già detto in Sezione 4, il nostro +/-Java non permette la definizione di sottoclassi e quindi non considera ereditarietà e overriding. Quindi, sebbene non utilizzate in queste note, si è preferito inserire questa informazione sul ritorno dei metodi come primo elemento della tripla in $\varphi_{m(C)}$ in modo da essere più vicini al Java ufficiale.

Inoltre, come in Java ufficiale, avremmo dovuto inserire anche l'informazione sul tipo dei parametri formali in ingresso ai metodi permettendo in $\varphi_{m(C)}$ associazioni del tipo $\langle \mathtt{T_r}, \mathtt{T_p\ x}, \mathtt{MB} \rangle /_{\mathtt{m}}$ con $\mathtt{T_p} \in \mathtt{TYPE\_NAME}$. Più precisamente, queste sono informazioni derivanti dall'analizzatore statico (e quindi dalla semantica statica dell'analisi statica da esso effettuata - vedi Sezione 3). Sono poi usate a run-time per selezionare correttamente il corpo del metodo giusto, per esempio, in caso di *overloading*[11]. Ovviamente, l'analizzatore statico controlla che non siano stati definiti due o più metodi identici per nome, tipo e numero di parametri.

[11]Nei linguaggi di programmazione, si dice *overloading* una famiglia di funzioni aventi lo *stesso nome*, ma con la possibilità di accettare *un diverso insieme di argomenti*, ed eventualmente restituire *un diverso valore di ritorno* (N.B.: in Java non viene considerato il valore di ritorno).

6.4 Il sistema *local_decl*

$$\begin{array}{lcl} \Gamma_{local\_decl} & = & \{\langle\, \texttt{VL},\, \sigma \,\rangle \mid \texttt{VL} \in \texttt{VAR\_DECL\_LIST},\ \sigma \in \texttt{State}\} \,\cup\, \{\sigma \mid \sigma \in \texttt{State}\} \\ T_{local\_decl} & = & \{\sigma \mid \sigma \in \texttt{State}\} \end{array}$$

$$\frac{\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho = \varphi{\cdot}\rho_1 \quad v = defval(\texttt{T}) \quad \rho' = \varphi[{}^{v}/_{\texttt{x}}]{\cdot}\rho_1 \quad \sigma' = \langle\, \rho',\, \mu,\, \delta \,\rangle}{\langle\, \texttt{T x;}\, , \sigma \,\rangle \longrightarrow_{local\_decl} \sigma'} \quad (local\_decl_{var})$$

$$\frac{\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \langle\, \texttt{E}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \quad \rho = \varphi{\cdot}\rho_1 \quad \rho' = \varphi[{}^{v}/_{\texttt{x}}]{\cdot}\rho_1 \quad \sigma' = \langle\, \rho',\, \mu_1,\, \delta \,\rangle}{\langle\, \texttt{T x = E;}\, , \sigma \,\rangle \longrightarrow_{local\_decl} \sigma'} \quad (local\_decl_{var\_init})$$

$$\frac{\texttt{V} \in \texttt{VAR\_DECL} \quad \langle\, \texttt{V}\, , \sigma \,\rangle \longrightarrow_{local\_decl} \sigma_1 \quad \langle\, \texttt{VLIST}\, , \sigma_1 \,\rangle \longrightarrow_{local\_decl} \sigma'}{\langle\, \texttt{V VLIST}\, , \sigma \,\rangle \longrightarrow_{local\_decl} \sigma'} \quad (local\_decl_{concat})$$

Il sistema ***local_decl*** formalizza la dichiarazione di variabili all'interno di metodi oppure blocchi. La regola $local\_decl_{var}$ formalizza la dichiarazione di una variabile `x` di tipo `T` senza inizializzazione; la regola $local\_decl_{var\_init}$ formalizza la dichiarazione di una variabile `x` di tipo `T` con inizializzazione contestuale alla dichiarazione. La dichiarazione di una variabile provoca la modifica della stack corrente ρ ed, in particolare, del suo frame più recente φ.

6.5 Il sistema *exp*

$$\begin{array}{lcl} \Gamma_{exp} & = & \{\langle\, \texttt{E},\, \sigma \,\rangle \mid \texttt{E} \in \texttt{EXP},\ \sigma \in \texttt{State}\} \cup \\ & & \{\langle\, v,\, \sigma \,\rangle \mid v \in Val^*\ ,\ \sigma \in \texttt{State}\} \\ T_{exp} & = & \{\langle\, v,\, \sigma \,\rangle \mid v \in Val^*\ ,\ \sigma \in \texttt{State}\} \end{array}$$

$$\frac{\eta(\texttt{c}) = v_c}{\langle\, \texttt{c}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v_c,\, \sigma \,\rangle} \quad (exp_{const})$$

$$\frac{\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{x}) = v_x}{\langle\, \texttt{x}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v_x,\, \sigma \,\rangle} \quad (exp_{ide})$$

$$\frac{\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{this}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \quad \varphi_{v(l)}(\texttt{x}) = v_x}{\langle\, \texttt{this.x}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_x,\, \sigma \,\rangle} \quad (exp_{ideobj})$$

$$\frac{\texttt{op} \in \texttt{OP} \quad \langle\, \texttt{E}_1\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_1,\, \sigma_1 \,\rangle \quad \langle\, \texttt{E}_2\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v_2,\, \sigma' \,\rangle \quad v = v_1\ op\ v_2}{\langle\, \texttt{E}_1\ \texttt{op}\ \texttt{E}_2\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{op}\ (metaregola))$$

$$\frac{\texttt{unop} \in \texttt{UNARYOP} \quad \langle\, \texttt{E}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad v_u = unop\ v}{\langle\, \texttt{unop}\ \texttt{E}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_u,\, \sigma' \,\rangle} \quad (exp_{unaryp}\ (metaregola))$$

$$\frac{\langle\, \texttt{E}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle}{\langle\, (\texttt{E})\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{()})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad l = newloc(\mu, \texttt{C}) \quad \delta(\texttt{C}) = \langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle \\ \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle = copy(\langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle, l) \quad \mu_1 = \mu[{}^{\langle\, \varphi_{v(l)}, \varphi_{m(C)} \,\rangle}/_{l}] \\ \varphi_{m(C)}(<\texttt{init}>) = \langle\, \texttt{void},\, \bot,\, \texttt{MB} \,\rangle \quad \texttt{MB} = \{\texttt{D}\ \texttt{S}\} \\ \sigma_{local} = \langle\, \omega[{}^{l}/_{\texttt{this}}],\, \mu_1,\, \delta \,\rangle \quad \langle\, \texttt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_2 \\ \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta \,\rangle \quad \langle\, \texttt{S}\,, \sigma_2 \,\rangle \longrightarrow_{com} \sigma_3 \\ \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_3,\, \delta \,\rangle\end{array}}{\langle\, \texttt{new C()}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, l,\, \sigma' \,\rangle} \quad (exp_{new\_nopar})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad l = newloc(\mu) \quad \delta(\texttt{C}) = \langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle \\ \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle = copy(\langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle, l) \quad \mu_1 = \mu[{}^{\langle\, \varphi_{v(l)}, \varphi_{m(C)} \,\rangle}/_{l}] \\ \varphi_{m(C)}(<\texttt{init}>) = \langle\, \texttt{void},\, \texttt{x},\, \texttt{MB} \,\rangle \quad \texttt{MB} = \{\texttt{D}\ \texttt{S}\} \\ \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \quad \langle\, \texttt{p}\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v_p,\, \sigma_2 \,\rangle \\ \sigma_2 = \langle\, \rho,\, \mu_2,\, \delta \,\rangle \quad \sigma_{local} = \langle\, \omega[{}^{v_p}/_{\texttt{x}}, {}^{l}/_{\texttt{this}}],\, \mu_2,\, \delta \,\rangle \\ \langle\, \texttt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_3 \quad \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta \,\rangle \\ \langle\, \texttt{S}\,, \sigma_3 \,\rangle \longrightarrow_{com} \sigma_4 \quad \sigma_4 = \langle\, \rho_4,\, \mu_4,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_4,\, \delta \,\rangle\end{array}}{\langle\, \texttt{new C(p)}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, l,\, \sigma' \,\rangle} \quad (exp_{new\_par})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad l = newloc(\mu) \quad \delta(\texttt{C}) = \langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle \\ \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle = copy(\langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle, l) \quad \mu_1 = \mu[{}^{\langle\, \varphi_{v(l)}, \varphi_{m(C)} \,\rangle}/_{l}] \\ \varphi_{m(C)}(<\texttt{init}>) = \bot \quad \sigma' = \langle\, \rho,\, \mu_1,\, \delta \,\rangle\end{array}}{\langle\, \texttt{new C()}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, l,\, \sigma' \,\rangle} \quad (exp_{new\_default})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{T}_\texttt{r},\, \bot,\, \texttt{MB} \,\rangle \quad \texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \quad \texttt{MB} = \{\texttt{D S return e;}\} \\ \sigma_{local} = \langle\, \omega[{}^{l}/_{\texttt{this}}],\, \mu,\, \delta \,\rangle \quad \langle\, \texttt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_1 \\ \sigma_1 = \langle\, \rho_1,\, \mu_1,\, \delta \,\rangle \quad \langle\, \texttt{S}\,, \sigma_1 \,\rangle \longrightarrow_{com} \sigma_2 \\ \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta \,\rangle \quad \langle\, \texttt{e}\,, \sigma_2 \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_3 \,\rangle \\ \sigma_3 = \langle\, \rho_2,\, \mu_3,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_3,\, \delta \,\rangle \end{array}}{\langle\, \texttt{obj.m()}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{mcall})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{T}_\texttt{r},\, \texttt{x},\, \texttt{MB} \,\rangle \quad \texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \quad \langle\, \texttt{p}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_p,\, \sigma_1 \,\rangle \\ \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \quad \texttt{MB} = \{\texttt{D S return e;}\} \\ \sigma_{local} = \langle\, \omega[{}^{v_p}/_{\texttt{x}},\, {}^{l}/_{\texttt{this}}],\, \mu_1,\, \delta \,\rangle \\ \langle\, \texttt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_2 \\ \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta \,\rangle \quad \langle\, \texttt{S}\,, \sigma_2 \,\rangle \longrightarrow_{com} \sigma_3 \\ \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta \,\rangle \quad \langle\, \texttt{e}\,, \sigma_3 \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_4 \,\rangle \\ \sigma_4 = \langle\, \rho_3,\, \mu_4,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_4,\, \delta \,\rangle \end{array}}{\langle\, \texttt{obj.m(p)}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{mcallpar})$$

Un'istanza della metaregola exp_{op}, per esempio nel caso in cui `op` sia `+`, è la seguente:

$$\frac{\texttt{+} \in \texttt{OP} \quad \langle\, \texttt{E}_1\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_1,\, \sigma_1 \,\rangle \quad \langle\, \texttt{E}_2\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v_2,\, \sigma' \,\rangle \quad v = v_1 \;+\; v_2}{\langle\, \texttt{E}_1 \texttt{ + } \texttt{E}_2\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{+})$$

Le configurazioni finali del sistema exp sono tutte del tipo $\langle v,\, \sigma \rangle$ con $v \in Val^*$. Siccome $\bot \notin Val^*$ il valore v di una espressione non può essere indefinito e quindi deve essere $v \neq \bot$. Per esempio, cosa accade con l'espressione 8/0?

Come già osservato in Sezione 6.3, nelle regole exp_{new_nopar}, exp_{new_par} e $exp_{new\_default}$ il costruttore è identificato in $\varphi_{m(C)}$ con `<init>` (si veda $\delta(<$ `init` $>) = \langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle$). Consistentemente con la JVM, questo permette alla nostra macchina astratta di riconoscere il corpo `MB` del costruttore, distinguerlo dagli altri metodi (anche quelli con lo stesso nome del costruttore) e, in luogo di una `new`, trattare la sua chiamata in modo speciale. Infatti, la JVM tratta il costruttore come un *metodo di inizializzazione di istanza*[12]. In luogo della creazione di un

[12]Sebbene in Java ufficiale esistano altri metodi di inizializzazione quali i *metodi di inizializzazione di classe e di interfaccia* (identificati con `<clinit>`), noi tratteremo so-

oggetto con il costrutto `new`, dopo la creazione di una nuova istanza della classe specificata (si veda Sezione 5.1), la nostra macchina risolve la chiamata al costruttore eseguendo il corpo del metodo `MB` associato ad `<init>`[13].

Nel caso in cui non vi siano dichiarazioni di variabili all'interno del corpo del metodo, le regole exp_{mcall} ed $exp_{mcallpar}$ diventano come riportato di seguito. Allo stesso modo posso essere facilmente modificate le regole exp_{new_nopar}, exp_{new_par} (fare per esercizio).

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{T}_\texttt{r},\, \bot,\, \texttt{MB} \,\rangle \quad \texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \quad \texttt{MB} = \{\texttt{S return e;}\} \\ \sigma_{local} = \langle\, \omega[{}^{l}/_{\texttt{this}}],\, \mu,\, \delta \,\rangle \quad \langle\, \texttt{S}\,, \sigma_{local} \,\rangle \longrightarrow_{com} \sigma_1 \\ \sigma_1 = \langle\, \rho_1,\, \mu_1,\, \delta \,\rangle \quad \langle\, \texttt{e}\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_2 \,\rangle \\ \sigma_2 = \langle\, \rho_1,\, \mu_2,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_2,\, \delta \,\rangle \end{array}}{\langle\, \texttt{obj.m()}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \; (exp_{mcallnodecl})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{T}_\texttt{r},\, \texttt{x},\, \texttt{MB} \,\rangle \quad \texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \quad \langle\, \texttt{p}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_p,\, \sigma_1 \,\rangle \\ \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \quad \texttt{MB} = \{\texttt{S return e;}\} \\ \sigma_{local} = \langle\, \omega[{}^{v_p}/_{\texttt{x}},\, {}^{l}/_{\texttt{this}}],\, \mu_1,\, \delta \,\rangle \\ \langle\, \texttt{S}\,, \sigma_{local} \,\rangle \longrightarrow_{com} \sigma_2 \\ \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta \,\rangle \quad \langle\, \texttt{e}\,, \sigma_2 \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_3 \,\rangle \\ \sigma_3 = \langle\, \rho_2,\, \mu_3,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_3,\, \delta \,\rangle \end{array}}{\langle\, \texttt{obj.m(p)}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \; (exp_{mcallparnodecl})$$

lo quelli che sono identificati con il nome speciale `<init>`, cioè i costruttori. A titolo informativo, i metodi di inizializzazione di classe vengono chiamati prima del costruttore.

[13]Per i più curiosi potrebbe essere interessante sapere che i metodi di inizializzazione di istanza come i costruttori sono invocati dalla JVM utilizzando un'istruzione bytecode "interna" chiamata *invokespecial*. Tali metodi sono chiamati una sola volta per ogni nuova istanza creata. Invece i metodi "standard" sono invocati dalla JVM utilizzando le istruzioni *invokevirtual* oppure *invokestatic*. Gli interessati possono approfondire questi aspetti consultando le specifiche ufficiali del linguaggio Java e della JVM.

6.6 Il sistema *com*

$$
\begin{array}{rcl}
\Gamma_{com} & = & \{\langle\, \mathtt{S},\, \sigma \,\rangle \mid \mathtt{S} \in \mathtt{STATEMENT},\ \sigma \in \mathtt{State}\} \cup \\
 & & \{\sigma \mid \sigma \in \mathtt{State}\} \\
T_{com} & = & \{\sigma \mid \sigma \in \mathtt{State}\}
\end{array}
$$

$$\frac{}{\langle\, \varepsilon\, , \sigma \,\rangle \longrightarrow_{com} \sigma} \quad (com_{empty})$$

$$\frac{}{\langle\, ;\, , \sigma \,\rangle \longrightarrow_{com} \sigma} \quad (com_{null})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \\ \langle\, \mathtt{E}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \\ \rho' = \rho[{}^{v}/_{\mathtt{x}}] \quad \sigma' = \langle\, \rho',\, \mu_1,\, \delta \,\rangle\end{array}}{\langle\, \mathtt{x} = \mathtt{E};\, , \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{assign})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \\ \langle\, \mathtt{E}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \\ \rho(\mathtt{this}) = l \quad \mu_1(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \quad \varphi'_{v(l)} = \varphi_{v(l)}[{}^{v}/_{\mathtt{x}}] \\ \mu_2 = \mu_1[{}^{\langle\, \varphi'_{v(l)},\, \varphi_{m(C)} \,\rangle}/_{l}] \quad \sigma' = \langle\, \rho,\, \mu_2,\, \delta \,\rangle\end{array}}{\langle\, \mathtt{this.x} = \mathtt{E};\, , \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{assign\_this})$$

$$\frac{\begin{array}{c}\mathtt{S} \in \mathtt{BASIC\_STATEMENT} \\ \langle\, \mathtt{S}\, , \sigma \,\rangle \longrightarrow_{com} \sigma_1 \quad \langle\, \mathtt{SLIST}\, , \sigma_1 \,\rangle \longrightarrow_{com} \sigma'\end{array}}{\langle\, \mathtt{S\ SLIST}\, , \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{concat})$$

$$\frac{\begin{array}{c}\langle\, \mathtt{E}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad v = \underline{true} \\ \langle\, \mathtt{S_1}\, , \sigma_1 \,\rangle \longrightarrow_{com} \sigma'\end{array}}{\langle\, \mathtt{if(E)\ S_1\ else\ S_2}\, , \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{if\_true})$$

$$\frac{\begin{array}{c}\langle\, \mathtt{E}\, , \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1 \,\rangle \quad v = \underline{false} \\ \langle\, \mathtt{S_2}\, , \sigma_1 \,\rangle \longrightarrow_{com} \sigma'\end{array}}{\langle\, \mathtt{if(E)\ S_1\ else\ S_2}\, , \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{if\_false})$$

$$\frac{\langle\, \texttt{E}\,, \sigma\,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_1\,\rangle \quad v = \underline{true} \qquad \langle\, \texttt{S}\,, \sigma_1\,\rangle \longrightarrow_{com} \sigma_2 \qquad \langle\, \texttt{while(E) S}\,, \sigma_2\,\rangle \longrightarrow_{com} \sigma'}{\langle\, \texttt{while(E) S}\,, \sigma\,\rangle \longrightarrow_{com} \sigma'} \quad (com_{while\_true})$$

$$\frac{\langle\, \texttt{E}\,, \sigma\,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma'\,\rangle \quad v = \underline{false}}{\langle\, \texttt{while(E) S}\,, \sigma\,\rangle \longrightarrow_{com} \sigma'} \quad (com_{while\_false})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta\,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)}\,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{void},\, \bot,\, \texttt{MB}\,\rangle \quad \texttt{MB} = \{\texttt{D S}\} \\ \sigma_{local} = \langle\, \omega[{}^{l}/_{\texttt{this}}],\, \mu,\, \delta\,\rangle \quad \langle\, \texttt{D}\,, \sigma_{local}\,\rangle \longrightarrow_{local\_decl} \sigma_1 \\ \sigma_1 = \langle\, \rho_1,\, \mu_1,\, \delta\,\rangle \quad \langle\, \texttt{S}\,, \sigma_1\,\rangle \longrightarrow_{com} \sigma_2 \\ \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta\,\rangle \quad \sigma' = \langle\, \rho,\, \mu_2,\, \delta\,\rangle\end{array}}{\langle\, \texttt{obj.m()}\,, \sigma\,\rangle \longrightarrow_{com} \sigma'} \quad (com_{mcall})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta\,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)}\,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{void, x, MB}\,\rangle \quad \langle\, \texttt{p}\,, \sigma\,\rangle \longrightarrow_{exp} \langle\, v_p,\, \sigma_1\,\rangle \\ \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta\,\rangle \quad \texttt{MB} = \{\texttt{D S}\} \\ \sigma_{local} = \langle\, \omega[{}^{v_p}/_{\texttt{x}},\, {}^{l}/_{\texttt{this}}],\, \mu_1,\, \delta\,\rangle \\ \langle\, \texttt{D}\,, \sigma_{local}\,\rangle \longrightarrow_{local\_decl} \sigma_2 \quad \sigma_2 = \langle\, \rho_2,\, \mu_2,\, \delta\,\rangle \\ \langle\, \texttt{S}\,, \sigma_2\,\rangle \longrightarrow_{com} \sigma_3 \quad \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta\,\rangle \\ \sigma' = \langle\, \rho,\, \mu_3,\, \delta\,\rangle\end{array}}{\langle\, \texttt{obj.m(p)}\,, \sigma\,\rangle \longrightarrow_{com} \sigma'} \quad (com_{mcallpar})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta\,\rangle \quad \rho_1 = push(\omega, \rho) \quad \sigma_1 = \langle\, \rho_1,\, \mu,\, \delta\,\rangle \\ \langle\, \texttt{D}\,, \sigma_1\,\rangle \longrightarrow_{local\_decl} \sigma_2 \quad \langle\, \texttt{S}\,, \sigma_2\,\rangle \longrightarrow_{com} \sigma_3 \\ \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta\,\rangle \quad \rho' = pop(\rho_3) \quad \sigma' = \langle\, \rho',\, \mu_3,\, \delta\,\rangle\end{array}}{\langle\, \{\texttt{D S}\}\,, \sigma\,\rangle \longrightarrow_{com} \sigma'} \quad (com_{block})$$

Nel sistema *com* la regola com_{empty} permette la trattazione di un insieme di comandi vuoto. Infatti, osservando la grammatica in Sezione 4, si nota che la categoria sintattica `STATEMENT` può produrre anche ε. Sono quindi ammissibili corpi di metodi e blocchi "vuoti". Ad esempio, `public void print(int n) {}` è un metodo valido e `{}` è un blocco valido. La regola com_{null} permette la trattazione del comando nullo `;` e il sistema di transizione si comporta alla stesso modo della regola com_{empty}, cioè non fa nulla. Facendo di nuovo un parallelo tra Java e il nostro +/- Java, queste due regole formalizzano il fatto che per un insieme di

comandi vuoto e per il comando nullo il compilatore Java non produce nessuna istruzione bytecode e che la JVM non ha azione da eseguire in merito.

Come già visto precedentemente, nelle regole com_{mcall} e $com_{mcallpar}$ il primo elemento della tripla $\varphi_{m(C)}(\mathtt{m}) = \langle\ \mathtt{void},\ \bot,\ \mathtt{MB}\ \rangle$ è uguale a `void` e questo indica che stiamo parlando del corpo di un metodo standard che non ha ritorno.

Con la regola com_{block} un blocco viene valutato in uno stato ottenuto impilando un nuovo frame vuoto ω sullo stack corrente ρ (si veda $\rho_1 = push(\omega, \rho)$). Tale frame conterrà poi le dichiarazioni di variabili locali al blocco. All'uscita del blocco, nello stato risultante dalla sua esecuzione, i nomi delle variabili locali (eventualmente modificate nel blocco) non esistono più (si veda $\rho' = pop(\rho_3)$ che rimuovere l'ultimo frame inserito) mentre permangono le eventuali modifiche apportate alle variabili precedentemente dichiarate all'esterno del blocco.

Si può osservare come la nostra semantica permetta la ridefinizione di una variabile `x` in un blocco annidato ad un blocco più esterno in cui `x` è stata precedentemente dichiarata. Una simile osservazione si può fare per la definizione di metodi che permettono la ridefinizione di una variabile all'interno dello stesso metodo. Invece, in Java, ciò non è possibile e, in entrambi i casi, si ha un errore statico in fase di compilazione. Nella nostra semantica operazionale, il problema delle dichiarazioni multiple della stessa variabile potrebbe essere risolto inserendo la precondizione $\rho(x) = \bot$ nelle regole $local\_decl_{var}$ e $local\_decl_{var\_init}$. Però , siccome questo errore in Java viene identificato staticamente, si è preferito non inserire tale controllo ma, verosimilmente, assumere che sia già stato eseguito a priori dal compilatore, mantenendo quindi la natura dinamica della nostra semantica.

$$\frac{\sigma = \langle\ \rho,\ \mu,\ \delta\ \rangle \quad \langle\ \mathtt{S}\ ,\sigma\ \rangle \longrightarrow_{com} \sigma' \quad \sigma' = \langle\ \rho',\ \mu',\ \delta\ \rangle}{\langle\ \{\mathtt{S}\}\ ,\sigma\ \rangle \longrightarrow_{com} \sigma'} \quad (com_{block\_nodecl})$$

Nel caso in cui nel blocco non vi siano dichiarazioni di variabili, la regola com_{block} diventa come sopra riportato. Allo stesso modo possono essere facilmente modificate le regole com_{mcall}, $com_{mcallpar}$ (fare per esercizio).

7 Esempi

7.1 Dichiarazione di una classe

```
class Rett {
  private int largh;
  private int alt;

  public int area() {
    return this.largh * this.alt;
  }

  public void scala(int sc) {
    this.largh = this.largh * sc;
    this.alt = this.alt * sc;
  }
}
```

La dichiarazione della precedente classe genera le seguenti strutture:

$\varphi_{v(Rett)}$:

largh	*defval*(int)
alt	*defval*(int)

$\varphi_{m(Rett)}$:

area	⟨ int, ⊥, {return this.largh * this.alt;} ⟩
scala	⟨ void, sc, { this.largh = this.largh * sc; this.alt = this.alt * sc; } ⟩

δ :

Rett	$\langle \varphi_{v(Rett)}, \varphi_{m(Rett)} \rangle$

7.2 Esecuzione di un programma

```
01:class Rett {
02:  private int largh;
03:  private int alt;
04:
05:  public void set_largh(int l) {
06:    this.largh = l;
07:  }
08:
09:  public void set_alt(int alt) {
10:    this.alt = alt;
11:  }
12:
13:  public int area() {
14:    return this.largh * this.alt;
15:  }
16:
17:  public void scala(int sc) {
18:    this.largh = this.largh * sc;
19:    this.alt = this.alt * sc;
20:  }
21:}
22:
23:public class Program {
24:  public static void main(String[] args) {
25:    Rett r = new Rett();
26:    int area = 0;
27:
28:    r.set_largh(3);
29:    r.set_alt(4);
30:    area = r.area();
31:  }
32:}
```

Partiamo dallo stato iniziale vuoto $\sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle$.

1) Applicazione della regola *prog* alle linee `01-32`

1.1) Applicazione della regola $decl_{class}$ alle linee `01-21`

1.1.1) Applicazione della regola cd_{concat} alle linee `02-20`

1.1.1.1) Applicazione della regola cd_{var} alla linea `02`

$\varphi_{v(Rett)}$:

`largh`	0

$\varphi_{m(Rett)}$: ω

1.1.1.2) Applicazione della regola cd_{var} alla linea `03`

$\varphi_{v(Rett)}$:

`largh`	0
`alt`	0

$\varphi_{m(Rett)}$: ω

1.1.1.3) Applicazione della regola cd_{meth} alle linee `05-07`

$\varphi_{v(Rett)}$:

`largh`	0
`alt`	0

$\varphi_{m(Rett)}$:

`set_largh`	$\langle$ `void, l, {this.largh = l;}` $\rangle$

1.1.1.4) Applicazione della regola cd_{meth} alle linee `09-11`

$\varphi_{v(Rett)}$:

`largh`	0
`alt`	0

$\varphi_{m(Rett)}$:

`set_largh`	$\langle$ `void, l, {this.largh = l;}` $\rangle$
`set_alt`	$\langle$ `void, alt, {this.alt = alt;}` $\rangle$

1.1.1.5) Applicazione della regola cd_{meth_nopar} alle linee `13-15`

$\varphi_{v(Rett)}$:

`largh`	0
`alt`	0

$\varphi_{m(Rett)}$:

`set_largh`	$\langle$ `void, l, {this.largh = l; }` $\rangle$
`set_alt`	$\langle$ `void, alt, {this.alt = alt; }` $\rangle$
`area`	$\langle$ `int,` $\perp$`, {return this.largh * this.alt;}` $\rangle$

1.1.1.6) Applicazione della regola cd_{meth} alle linee `17-20`

$\varphi_{v(Rett)}$:

`largh`	0
`alt`	0

$\varphi_{m(Rett)}$:

`set_largh`	⟨ `void, l, {this.largh = l; }` ⟩
`set_alt`	⟨ `void, alt, {this.alt = alt; }` ⟩
`area`	⟨ `int,` ⊥, `{return this.largh * this.alt;}` ⟩
`scala`	⟨ `void, sc, {this.largh = this.largh * sc; this.alt = this.alt * sc; }` ⟩

1.1) Al termine dell'applicazione di $decl_{class}$ lo stato è il seguente:

$$\sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle$$

δ_1 :

`Rett`	⟨ $\varphi_{v(Rett)}$, $\varphi_{m(Rett)}$ ⟩

1.2) Binding del parametro del metodo `main` alla linea `24`

$$\sigma_2 = \langle\, \rho_1,\, \mu_0,\, \delta_1 \,\rangle$$

ρ_1 :

`args`	*null*

1.3) Applicazione della regola $local\_decl_{concat}$ alle linee `25-26`

1.3.1) Applicazione della regola $local\_decl_{var\_init}$ alla linea `25`

1.3.1.1) Applicazione della regola $exp_{new\_default}$ alla linea `25`

$$\sigma_3 = \langle\, \rho_2,\, \mu_1,\, \delta_1 \,\rangle$$

ρ_2 :

`args`	*null*
`r`	l_1

μ_1 :

l_1	⟨ $\varphi_{v(l_1)}$: `largh` 0 / `alt` 0 , $\varphi_{m(Rett)}$ ⟩

1.3.2) Applicazione della regola $local\_decl_{var\_init}$ alla linea `26`

1.3.2.1) Applicazione di exp_{const} alla linea `26`

1.3.2) Al termine dell'applicazione di $local\_decl_{var\_init}$ lo stato è il seguente:

$$\sigma_4 = \langle\, \rho_3,\, \mu_1,\, \delta_1 \,\rangle$$

ρ_3 :

`args`	*null*
`r`	l_1
`area`	0

μ_1 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`largh` | 0 `alt` | 0	, $\varphi_{m(Rett)}$ $\rangle$

1.4) Applicazione della regola com_{concat} alle linee `28-30`

1.4.1) Applicazione della regola $com_{mcallpar}$ alla linea `28`. Lo stato risultante da tale applicazione è il seguente:

$$\sigma_5 = \langle\, \rho_3,\, \mu_2,\, \delta_1 \,\rangle$$

μ_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`largh` | 3 `alt` | 0	, $\varphi_{m(Rett)}$ $\rangle$

1.4.2) Applicazione della regola $com_{mcallpar}$ alla linea `29`. Lo stato risultante da tale applicazione è il seguente:

$$\sigma_6 = \langle\, \rho_3,\, \mu_3,\, \delta_1 \,\rangle$$

μ_3 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`largh` | 3 `alt` | 4	, $\varphi_{m(Rett)}$ $\rangle$

1.4.3) Applicazione della regola com_{assign} alla linea `30`

1.4.4.1) Applicazione della regola exp_{mcall} alla linea `30`

1.4.4) Al termine dell'applicazione di com_{assign} lo stato è il seguente:

$$\sigma_7 = \langle\, \rho_4,\, \mu_3,\, \delta_1 \,\rangle$$

ρ_4 :

`args`	*null*
`r`	l_1
`area`	12

1.4) Al termine dell'applicazione di com_{concat} lo stato è il seguente:

$$\sigma_7 = \langle\, \rho_4,\, \mu_3,\, \delta_1 \,\rangle$$

1) Al termine dell'applicazione di *prog* lo stato è il seguente:

$$\sigma_7 = \langle\, \rho_4,\, \mu_3,\, \delta_1 \,\rangle$$

ρ_4 :

`args`	*null*
`r`	l_1
`area`	12

μ_3 :

l_1	$\langle\ \varphi_{v(l_1)}$: `largh` 3, `alt` 4 , $\varphi_{m(Rett)}\,\rangle$

δ_1 :

`Rett`	$\langle\, \varphi_{v(Rett)},\, \varphi_{m(Rett)}\,\rangle$

7.3 Swap

```
01:class Coord {
02:  private int x;
03:  private int y;
04:
05:  public void set_prima_coordinata(int l) {
06:    this.x = l;
07:  }
08:
09:  public void set_seconda_coordinata(int a) {
10:    this.y = a;
11:  }
12:
13:  public int get_prima_coordinata() {
14:    return this.x;
15:  }
16:
17:  public int get_seconda_coordinata() {
18:    return this.y;
19:  }
20:}
21:
22:class Swap {
23:  public void swap(Coord a, Coord b) {
24:    Coord temp = new Coord();
25:
26:    temp = a;
27:    a = b;
28:    b = temp;
29:  }
30:}
31:
32:public class Program {
33:  public static void main(String[] args) {
34:    Coord obj1 = new Coord();
35:    Coord obj2 = new Coord();
36:    Swap s = new Swap();
37:    int t = 0;
38:
39:    obj1.set_prima_coordinata(10);
40:    obj1.set_seconda_coordinata(20);
41:    obj2.set_prima_coordinata(80);
42:    obj2.set_seconda_coordinata(90);
```

```
43:
44:     s.swap(obj1, obj2);
45:     t = obj1.get_prima_coordinata();
46:  }
47:}
```

Per valutare la semantica di questo programma abbiamo bisogno di una semplice estensione delle regole dei sistemi *cd* e *com*, in particolare delle regole per la dichiarazione di metodi e per la loro invocazione. Questa estensione è necessaria per gestire il metodo `swap` della classe `Swap` che necessita di due parametri:

$$\frac{\varphi'_{m(C)} = \varphi_{m(C)}[\,\langle\, \mathtt{T_r}, \mathtt{x_1}, \mathtt{x_2}, \mathtt{MB} \,\rangle / \mathtt{m}\,]}{\langle\, \mathtt{public\ T_r\ m(T_{p1}\ x_1,\ T_{p2}\ x_2)\ MB}\,, \langle\, \varphi_{v(C)},\, \varphi_{m(C)} \,\rangle \,\rangle \longrightarrow_{cd} \langle\, \varphi_{v(C)},\, \varphi'_{m(C)} \,\rangle} \quad (cd_{meth2})$$

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\mathtt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\mathtt{m}) = \langle\, \mathtt{void},\, \mathtt{x_1},\, \mathtt{x_2},\, \mathtt{MB} \,\rangle \\ \langle\, \mathtt{p_1}\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_{p1},\, \sigma_1 \,\rangle \quad \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \\ \langle\, \mathtt{p_2}\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v_{p2},\, \sigma_2 \,\rangle \quad \sigma_2 = \langle\, \rho,\, \mu_2,\, \delta \,\rangle \\ \mathtt{MB} = \{\mathtt{D\ S}\} \quad \sigma_{local} = \langle\, \omega[\,{}^{v_{p1}}/_{\mathtt{x_1}},\, {}^{v_{p2}}/_{\mathtt{x_2}},\, {}^{l}/_{\mathtt{this}}\,],\, \mu_2,\, \delta \,\rangle \\ \langle\, \mathtt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_3 \\ \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta \,\rangle \quad \langle\, \mathtt{S}\,, \sigma_3 \,\rangle \longrightarrow_{com} \sigma_4 \\ \sigma_4 = \langle\, \rho_4,\, \mu_4,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_4,\, \delta \,\rangle \end{array}}{\langle\, \mathtt{obj.m(p_1, p_2)}\,, \sigma \,\rangle \longrightarrow_{com} \sigma'} \quad (com_{mcallpar2})$$

Il sistema *exp* può essere esteso per valutare la chiamata di metodi con due parametri in modo simile:

$$\frac{\begin{array}{c}\sigma = \langle\, \rho,\, \mu,\, \delta \,\rangle \quad \rho(\texttt{obj}) = l \quad \mu(l) = \langle\, \varphi_{v(l)},\, \varphi_{m(C)} \,\rangle \\ \varphi_{m(C)}(\texttt{m}) = \langle\, \texttt{T}_\texttt{r},\, \texttt{x}_1,\, \texttt{x}_2,\, \texttt{MB} \,\rangle \\ \texttt{T}_\texttt{r} \in \texttt{TYPE\_NAME} \quad \langle\, \texttt{p}_1\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v_{p1},\, \sigma_1 \,\rangle \quad \sigma_1 = \langle\, \rho,\, \mu_1,\, \delta \,\rangle \\ \langle\, \texttt{p}_2\,, \sigma_1 \,\rangle \longrightarrow_{exp} \langle\, v_{p2},\, \sigma_2 \,\rangle \quad \sigma_2 = \langle\, \rho,\, \mu_2,\, \delta \,\rangle \\ \texttt{MB} = \{\texttt{D S return e;}\} \quad \sigma_{local} = \langle\, \omega[{}^{v_{p1}}/_{\texttt{x}_1},\, {}^{v_{p2}}/_{\texttt{x}_2},\, {}^{l}/_{\texttt{this}}],\, \mu_2,\, \delta \,\rangle \\ \langle\, \texttt{D}\,, \sigma_{local} \,\rangle \longrightarrow_{local\_decl} \sigma_3 \\ \sigma_3 = \langle\, \rho_3,\, \mu_3,\, \delta \,\rangle \quad \langle\, \texttt{S}\,, \sigma_3 \,\rangle \longrightarrow_{com} \sigma_4 \\ \sigma_4 = \langle\, \rho_4,\, \mu_4,\, \delta \,\rangle \quad \langle\, \texttt{e}\,, \sigma_4 \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma_5 \,\rangle \\ \sigma_5 = \langle\, \rho_4,\, \mu_5,\, \delta \,\rangle \quad \sigma' = \langle\, \rho,\, \mu_5,\, \delta \,\rangle \end{array}}{\langle\, \texttt{obj.m}(\texttt{p}_1, \texttt{p}_2)\,, \sigma \,\rangle \longrightarrow_{exp} \langle\, v,\, \sigma' \,\rangle} \quad (exp_{mcallpar2})$$

Partiamo dallo stato iniziale vuoto $\sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle$.

1) Applicazione della regola *prog* alle linee `01-47`

1.1) Applicazione della regola $decl_{concat}$ alle linee `01-30` Al termine dell'applicazione di questa regola lo stato è il seguente:

$\varphi_{v(Coord)}$:

x	0
y	0

$\varphi_{m(Coord)}$:

set_prima_coordinata	⟨ void, l, {this.x = l; } ⟩
set_seconda_coordinata	⟨ void, a, {this.y = a; } ⟩
get_prima_coordinata	⟨ int, ⊥, {return this.x;} ⟩
get_seconda_coordinata	⟨ int, ⊥, {return this.y;} ⟩

$\varphi_{v(Swap)}$: ω

$\varphi_{m(Swap)}$:

swap	⟨ void, a, b, {Coord temp = new Coord(); temp = a; a = b; b = temp; } ⟩

$$\sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle$$

δ_1 :

Coord	$\langle\, \varphi_{v(Coord)},\, \varphi_{m(Coord)} \,\rangle$
Swap	$\langle\, \varphi_{v(Swap)},\, \varphi_{m(Swap)} \,\rangle$

1.2) Binding del parametro del metodo `main` ed applicazione delle regole del sistema *local_decl* alle linee `34-37`

$$\sigma_2 = \langle\, \rho_1,\, \mu_1,\, \delta_1 \,\rangle$$

ρ_1 :

`args`	*null*
`obj1`	l_1
`obj2`	l_2
`s`	l_3
`t`	0

μ_1 :

l_1	$\langle\ \varphi_{v(l_1)}$: `x` 0, `y` 0 , $\varphi_{m(Coord)}\ \rangle$
l_2	$\langle\ \varphi_{v(l_2)}$: `x` 0, `y` 0 , $\varphi_{m(Coord)}\ \rangle$
l_3	$\langle\ \varphi_{v(l_3)} : \omega\ ,\ \varphi_{m(Swap)}\ \rangle$

1.3) Applicazione di com_{concat} alle linee `39-44`

1.3.1) Applicazione di $com_{mcallpar}$ alla linea `39`

1.3.2) Applicazione di $com_{mcallpar}$ alla linea `40`

1.3.3) Applicazione di $com_{mcallpar}$ alla linea `41`

1.3.4) Applicazione di $com_{mcallpar}$ alla linea `42`
Al termine dell'applicazione di questa regola lo stato è il seguente:

$$\sigma_3 = \langle\, \rho_1,\, \mu_2,\, \delta_1 \,\rangle$$

ρ_1 :

`args`	*null*
`obj1`	l_1
`obj2`	l_2
`s`	l_3
`t`	0

μ_2 :

l_1	$\langle\ \varphi_{v(l_1)}$: `x` 10, `y` 20 , $\varphi_{m(Coord)}\ \rangle$
l_2	$\langle\ \varphi_{v(l_2)}$: `x` 80, `y` 90 , $\varphi_{m(Coord)}\ \rangle$
l_3	$\langle\ \varphi_{v(l_3)} : \omega\ ,\ \varphi_{m(Swap)}\ \rangle$

1.3.5) Applicazione di $com_{mcallpar2}$ alla linea `44`

$$\rho_1(\texttt{s}) = l_3$$
$$\mu_2(l_3) = \langle\ \varphi_{v(l_3)},\ \varphi_{m(Swap)}\ \rangle$$
$$\sigma_{local1} = \langle\ \rho_{local1},\ \mu_2,\ \delta_1\ \rangle$$

ρ_{local1} :

`a`	l_1
`b`	l_2
`this`	l_3

μ_2 :

l_1	$\langle\ \varphi_{v(l_1)}$: `x` 10, `y` 20 , $\varphi_{m(Coord)}\ \rangle$
l_2	$\langle\ \varphi_{v(l_2)}$: `x` 80, `y` 90 , $\varphi_{m(Coord)}\ \rangle$
l_3	$\langle\ \varphi_{v(l_3)} : \omega\ ,\ \varphi_{m(Swap)}\ \rangle$

Osservazione: in σ_{local1} i valori delle locazioni l_1 e l_2 sono stati semplicemente copiati nelle variabili locali `a` e `b`, rispettivamente.

1.3.5.1) Applicazione di $local\_decl_{var\_init}$ alla linea `24`

$$\sigma_{local2} = \langle\ \rho_{local2},\ \mu_3,\ \delta_1\ \rangle$$

ρ_{local2} :

`a`	l_1
`b`	l_2
`this`	l_3
`temp`	l_4

μ_3 :

l_1	$\langle \ \varphi_{v(l_1)}$: `x` 10, `y` 20 , $\varphi_{m(Coord)} \rangle$
l_2	$\langle \ \varphi_{v(l_2)}$: `x` 80, `y` 90 , $\varphi_{m(Coord)} \rangle$
l_3	$\langle \ \varphi_{v(l_3)} : \omega \ , \ \varphi_{m(Swap)} \rangle$
l_4	$\langle \ \varphi_{v(l_4)}$: `x` 0, `y` 0 , $\varphi_{m(Coord)} \rangle$

1.3.5.2) Applicazione di com_{concat} alle linee `26-28`

$$\sigma_{local3} = \langle \ \rho_{local3}, \ \mu_3, \ \delta_1 \ \rangle$$

ρ_{local3} :

`a`	l_2
`b`	l_1
`this`	l_3
`temp`	l_1

μ_3 :

l_1	$\langle \ \varphi_{v(l_1)}$: `x` 10, `y` 20 , $\varphi_{m(Coord)} \rangle$
l_2	$\langle \ \varphi_{v(l_2)}$: `x` 80, `y` 90 , $\varphi_{m(Coord)} \rangle$
l_3	$\langle \ \varphi_{v(l_3)} : \omega \ , \ \varphi_{m(Swap)} \rangle$
l_4	$\langle \ \varphi_{v(l_4)}$: `x` 0, `y` 0 , $\varphi_{m(Coord)} \rangle$

1.3.5) Al termine dell'applicazione della regola $com_{mcallpar2}$ lo stato è il seguente:

$$\sigma_4 = \langle\, \rho_1,\, \mu_3,\, \delta_1 \,\rangle$$

ρ_1 :

`args`	*null*
`obj1`	l_1
`obj2`	l_2
`s`	l_3
`t`	0

μ_3 :

l_1	$\langle\ \varphi_{v(l_1)}$: `x` 10, `y` 20 , $\varphi_{m(Coord)}\ \rangle$
l_2	$\langle\ \varphi_{v(l_2)}$: `x` 80, `y` 90 , $\varphi_{m(Coord)}\ \rangle$
l_3	$\langle\, \varphi_{v(l_3)} : \omega\ ,\ \varphi_{m(Swap)}\, \rangle$
l_4	$\langle\ \varphi_{v(l_4)}$: `x` 0, `y` 0 , $\varphi_{m(Coord)}\ \rangle$

Applicare la regola com_{assign} alla linea `45` e dire quanto vale la variabile `t` dopo l'applicazione. Abbiamo realizzato effettivamente lo swap oppure no? ....

7.3.1 Swap effettivo

Per realizzare lo swap tra le due istanze dobbiamo modificare il metodo `swap` della classe `Swap` nel seguente modo:

```
...
class Swap {
  public void swap(Coord a, Coord b) {
    Coord temp = new Coord();

    temp.set_prima_coordinata(a.get_prima_coordinata());
    temp.set_seconda_coordinata(a.get_seconda_coordinata());

    a.set_prima_coordinata(b.get_prima_coordinata());
    a.set_seconda_coordinata(b.get_seconda_coordinata());
```

```
    b.set_prima_coordinata(temp.get_prima_coordinata());
    b.set_seconda_coordinata(temp.get_seconda_coordinata());
  }
}
...
```

Verificare per esercizio che con la precedente modifica si realizza effettivamente lo swap. Cosa accade se modifichiamo i metodi della classe `Coord` nel modo seguente?

```
 ...
04:
05:  public void set_prima_coordinata(int x) {
06:    this.x = x;
07:  }
08:
09:  public void set_seconda_coordinata(int y) {
10:    this.y = y;
11:  }
 ...
```

7.4 Fattoriale

```
00:class Fact {
01:  private int f;
02:  private int prof;
03:
04:  public int fact(int n) {
05:      this.prof = this.prof + 1;
06:      if (n == 0)
07:          this.f = 1;
08:      else
09:          this.f = n * this.fact(n-1);
10:
11:     return this.f;
12:  }
13:}
14:
15:public class Program {
16:  public static void main(String[] foo) {
17:      Fact fact = new Fact();
18:      int res = 0;
19:
20:      res = fact.fact(2);
21:  }
22:}
```

Osservazione: Come già osservato in Sezione 4.2 il nostro +/-Java obbliga un metodo non `void` ad avere il `return` alla fine del suo corpo. Senza questa restrizione avremmo potuto scrivere le linee di codice `06-11` in questo modo:

```
...
if (n == 0)
    return 1;
else
    return n * fact(n-1);
...
```

Inoltre, essendo all'interno della definizione di una classe, mentre in Java avremmo potuto richiamare la funzione `fact` scrivendo semplicemente `n * fact(n-1)`, in +/-Java dobbiamo usare il `this` e scrivere quindi `n * this.fact(n-1)`.

Partiamo dallo stato iniziale vuoto $\sigma_0 = \langle\, \rho_0,\, \mu_0,\, \delta_0 \,\rangle$.

1) Applicazione della regola *prog* alle linee `00-22`

1.1) Applicazione della regola $decl_{class}$ alle linee `00-13` Al termine dell'applicazione di questa regola lo stato è il seguente:

$\varphi_{v(Fact)}$:

`f`	0
`prof`	0

$\varphi_{m(Fact)}$:

`fact`	$\langle$ `int, n,`	`{ this.prof = this.prof + 1;`
		`if (n == 0) this.f = 1;`
		`else this.f = n * fact(n-1);`
		`return this.f; }` $\rangle$

$$\sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle$$

δ_1 :

`Fact`	$\langle\, \varphi_{v(Fact)},\, \varphi_{m(Fact)} \,\rangle$

Come sarà chiaro di seguito, la variabile di istanza `prof` indicherà sia il numero di chiamate ricorsive alla funzione `fact` sia la profondità del livello di annidamento di tali chiamate.

1.2) Binding del parametro del metodo `main` ed applicazione delle regole del sistema *local_decl* alle linee `17-18`

$$\sigma_2 = \langle\, \rho_1,\, \mu_1,\, \delta_1 \,\rangle$$

ρ_1 :

`foo`	*null*
`fact`	l_1
`res`	0

μ_1 :

l_1	$\langle\ \varphi_{v(l_1)}$: `f` 0 / `prof` 0 , $\varphi_{m(Fact)}\ \rangle$

1.3) Applicazione di com_{assign} alla linea `20`: `res = fact.fact(2);`

1.3.1) Prima applicazione di $exp_{mcallparnodecl}$ alla linea `20` su `fact.fact(2);`

$$\rho_1(\texttt{fact}) = l_1$$
$$\mu_1(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fact)} \,\rangle$$
$$\sigma_{local1} = \langle\, \rho_{local1},\, \mu_1,\, \delta_1 \,\rangle$$

ρ_{local1} :

`n`	2
`this`	l_1

μ_1 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 0 `prof` | 0	, $\varphi_{m(Fact)}$ $\rangle$

1.3.2) Applicazione di com_{concat} alle linee `05-09`. Applicazione di com_{assign_this} alla linea `05`. Applicazione di com_{if_false} alle linee `06-09` e quindi applicazione di com_{assign_this} alla linea `09`: `this.f = 2 * this.fact(1);` (ACTIVE).

$$\sigma'_{local1} = \langle\, \rho_{local1},\, \mu_2,\, \delta_1 \,\rangle$$

ρ_{local1} :

`n`	2
`this`	l_1

μ_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 0 `prof` | 1	, $\varphi_{m(Fact)}$ $\rangle$

Come fatto negli esempi precedenti, μ_2 dovrebbe rappresentare l'heap μ_1 dopo che `f` è stato modificato dall'assegnamento `this.f = 2 * this.fact(1);`. Però, in questo caso, abbiamo che il lato destro dell'assegnamento è una chiamata ricorsiva alla funzione `this.fact(1)` e quindi l'assegnamento non può terminare ma deve attendere la terminazione di tale chiamata ricorsiva. Per questo motivo lo stack locale ρ_{local1} rimane attivo (la scritta ACTIVE sopra indica proprio questa condizione) e l'heap μ_2 non riporta subito la modifica su `f` che per ora rimane a zero. Quindi, si applica di nuovo la regola $exp_{mcallparnodecl}$ a `this.fact(1);` istanziando `obj` con `this` nella parte sinistra della regola stessa e considerando che lo stack correntemente attivo è ora ρ_{local1}. Si noti che in memoria abbiamo ora due stack ρ e ρ_{local1}, ma solo uno di loro è attivo e cioè ρ_{local1}.

1.3.2.1) Seconda applicazione di $exp_{mcallparnodecl}$ alla linea `09` su `this.fact(1);`

$$\rho_{local1}(\texttt{this}) = l_1$$
$$\mu_2(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fact)} \,\rangle$$
$$\sigma_{local2} = \langle\, \rho_{local2},\, \mu_2,\, \delta_1 \,\rangle$$

ρ_{local2} :

`n`	1
`this`	l_1

μ_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f`	0	, $\varphi_{m(Fact)}$ $\rangle$
			`prof`	1	

1.3.2.2) Applicazione di com_{concat} alle linee `05-09`. Applicazione di com_{assign_this} alla linea `05`. Applicazione di com_{if_false} alle linee `06-09` e quindi applicazione di com_{assign_this} alla linea `09`: `this.f = 1 * this.fact(0);` (ACTIVE).

$$\sigma'_{local2} = \langle\, \rho_{local2},\, \mu_3,\, \delta_1 \,\rangle$$

ρ_{local2} :

`n`	1
`this`	l_1

μ_3 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f`	0	, $\varphi_{m(Fact)}$ $\rangle$
			`prof`	2	

Come per il punto 1.3.2, μ_3 dovrebbe rappresentare l'heap μ_2 dopo che `f` è stato modificato dal secondo assegnamento `this.f = 1 * this.fact(0);`. Però, ancora un volta, abbiamo che il lato destro dell'assegnamento è una ulteriore chiamata ricorsiva alla funzione `this.fact(0)` e quindi l'assegnamento non può terminare ma deve attendere la terminazione di tale chiamata ricorsiva. Lo stack locale ρ_{local2} rimane quindi attivo e l'heap μ_3 non riporta subito la modifica su `f` che rimane ancora a zero. Quindi, all'interno della regola com_{assign_this} alla linea `09`, si applica di nuovo la regola $exp_{mcallparnodecl}$ a `this.fact(0);` considerando che lo stack correntemente attivo è ora ρ_{local2}. In memoria abbiamo ora tre stack ρ, ρ_{local1} e ρ_{local2}, ma solo uno di loro è attivo e cioè ρ_{local2}.

1.3.2.2.1) Terza applicazione di $exp_{mcallparnodecl}$ alla linea `09` su `this.fact(0);`

$$\rho_{local2}(\texttt{this}) = l_1$$
$$\mu_3(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fact)} \,\rangle$$
$$\sigma_{local3} = \langle\, \rho_{local3},\, \mu_3,\, \delta_1 \,\rangle$$

ρ_{local3} :

`n`	0
`this`	l_1

μ_3 :

l_1	$\langle \quad \varphi_{v(l_1)}$:	`f`	0	, $\varphi_{m(Fact)} \rangle$
		`prof`	2	

In memoria abbiamo quindi quattro stack ρ, ρ_{local1}, ρ_{local2} e ρ_{local3}, ma solo uno di loro è attivo e cioè ρ_{local3}.

1.3.2.2.2) Applicazione di com_{concat} alle linee `05-09`. Applicazione di com_{assign_this} alla linea `05`. Applicazione di com_{if_true} alle linee `06-09` e quindi applicazione di com_{assign_this} alla linea `07`: `this.f = 1;`.

Ora, a differenza di prima, si è scelto il ramo *true* dell'`if` e l'assegnamento `this.f = 1;` alla linea `07` può essere completato con la regola com_{assign_this}. Si può quindi passare alla linea `11` ad eseguire il `return` finale della terza chiamata ricorsiva. Quindi, σ'_{local3} di seguito riportato è lo stato locale a livello di profondità 3 dopo la valutazione dello statement `return this.f` (cioè `return 1`).

$$\sigma'_{local3} = \langle\, \rho_{local3},\, \mu_4,\, \delta_1 \,\rangle$$

ρ_{local3} :

`n`	0
`this`	l_1

μ_4 :

l_1	$\langle \quad \varphi_{v(l_1)}$:	`f`	1	, $\varphi_{m(Fact)} \rangle$
		`prof`	3	

Dopo questo `return`, essendosi conclusa la terza chiamata a `this.fact(0);` del punto 1.3.2.2.1, si può risalire al livello 2. A tale livello, l'assegnamento `this.f = 1 * this.fact(0);` (e cioè `this.f = 1 * 1;`) nel punto 1.3.2.2 può ora essere effettuato e lo stato locale risultante è:

$$\sigma''_{local2} = \langle\, \rho_{local2},\, \mu'_3,\, \delta_1 \,\rangle$$

ρ_{local2} :

n	1
this	l_1

μ'_3 :

l_1	$\langle \ \varphi_{v(l_1)}$:	f	1	, $\varphi_{m(Fact)} \ \rangle$
		prof	3	

Subito dopo il `return` della terza chiamata ricorsiva, lo stack locale ρ_{local3} in σ'_{local3} non serve più e può essere cancellato dalla memoria[14].

Invece, osservando lo stato σ''_{local2} del livello due, si nota che lo stack locale a tale livello è ancora ρ_{local2} mentre l'heap μ'_3 riporta le modifiche sia della variabile d'istanza `f` sia della variabile d'istanza `prof`. Quindi risalendo le modifiche apportate all'heap "nei livelli inferiori" permangono.

Si può quindi passare di nuovo alla linea `11` ad eseguire il `return` finale della seconda chiamata ricorsiva.

Dopo questo `return`, essendosi conclusa anche la seconda chiamata a `this.fact(1);` del punto 1.3.2.1, si può risalire al livello 1. A tale livello, l'assegnamento `this.f = 2 * this.fact(1);` (e cioè `this.f = 2 * 1;`) nel punto 1.3.2 può ora essere effettuato e lo stato locale risultante è:

$$\sigma''_{local1} = \langle \, \rho_{local1}, \, \mu'_2, \, \delta_1 \, \rangle$$

ρ_{local1} :

n	2
this	l_1

μ'_2 :

l_1	$\langle \ \varphi_{v(l_1)}$:	f	2	, $\varphi_{m(Fact)} \ \rangle$
		prof	3	

Da questo momento in poi, lo stack locale ρ_{local2} in σ''_{local2} non serve più e può essere cancellato dalla memoria.

Invece, nello stato σ''_{local1} del livello 1, lo stack locale a tale livello è ancora ρ_{local1} mentre l'heap μ'_2 riporta le modifiche sia della variabile d'istanza `f` sia della variabile d'istanza `prof`. Quindi, come è ovvio, risalendo ulteriormente le modifiche apportate all'heap dai livelli inferiori ancora permangono.

Si può quindi passare di nuovo alla linea `11` ad eseguire il `return` finale della prima chiamata ricorsiva.

[14] Questa operazione di cancellazione in Java è effettuata dal *garbage collector* non preso in considerazione in queste note.

Dopo questo `return`, essendosi conclusa la prima chiamata a `this.fact(2);` del punto 1.3.1, si può risalire al livello 0. A tale livello, l'assegnamento `res = fact.fact(2);` (e cioè `res = 2;`) nel punto 1.3 può ora essere effettuato e lo stato finale prima di concludere la funzione `main` è:

$$\sigma_3 = \langle\, \rho'_1,\, \mu'_1,\, \delta_1 \,\rangle$$

ρ'_1 :

`foo`	*null*
`fact`	l_1
`res`	2

μ'_1 :

l_1	$\langle\ \varphi_{v(l_1)}$: `f` 2, `prof` 3 , $\varphi_{m(Fact)}\ \rangle$

Da questo momento in poi, lo stack locale ρ_{local1} in σ''_{local1} non serve più e può essere cancellato dalla memoria. Per evidenziare le "discese" e le "risalite" delle chiamate (rispetto agli indici usati per lo stack) si può scrivere la sequenza in questo modo: $\rho_0 \longrightarrow \rho_1 \searrow \rho_{local1} \searrow \rho_{local2} \searrow \rho_{local3} \nearrow \rho_{local2} \nearrow \rho_{local1} \nearrow \rho'_1$.

Si noti che μ'_3, μ'_2 e μ'_1, con la loro numerazione decrescente, rappresentano i vari heap dei differenti livelli durante il "processo di risalita" delle chiamate ricorsive. In realtà, l'heap μ'_1 è uguale all'heap μ'_2 in quanto al livello della funzione `main` non si effettuano ulteriori modifiche all'heap ma si assegna solo la variabile `res` nello stack. Si è preferito usare il pedice "$_1$" in μ'_1 per indicare che si risaliti ad un livello superiore. Pertanto, deve sempre essere chiaro che la sequenza μ_0, μ_1, μ_2, μ_3, μ_4, μ'_3, μ'_2, μ'_1 indica modifiche successive sempre allo stesso heap ed in particolare sempre alla stessa istanza `fact` riferita nell'heap da l_1. Per evidenziare le discese e le risalite delle chiamate (rispetto agli indici usati per l'heap) si può scrivere la sequenza in questo modo: $\mu_0 \longrightarrow \mu_1 \searrow \mu_2 \searrow \mu_3 \searrow \mu_4 \nearrow \mu'_3 \nearrow \mu'_2 \nearrow \mu'_1$.

7.5 Fibonacci

```
01:class Fib {
02:
03:    private int f;
04:    private int count;
05:
06:
07:    public int fibonacci(int n) {
08:
09:        this.count = this.count + 1;
10:
11:
12:        if (n == 0)
13:            this.f = 0;
14:        else
15:            if (n == 1)
16:                this.f = 1;
17:            else
18:                this.f = this.fibonacci(n - 1) + this.fibonacci(n - 2);
19:
20:
21:        return this.f;
22:    }
23:}
24:
25:public class Program {
26:  public static void main(String[] foo) {
27:
28:        Fib fib = new Fib();
29:        int res = 0;
30:
31:        res = fib.fibonacci(2);
32:  }
33:}
```

Partiamo dallo stato iniziale vuoto $\sigma_0 = \langle\, \rho_0, \mu_0, \delta_0 \,\rangle$.

1) Applicazione della regola *prog* alle linee `01-33`

1.1) Applicazione della regola $decl_{class}$ alle linee `01-23` Al termine dell'applicazione di questa regola lo stato è il seguente:

$\varphi_{v(Fib)}$:

`f`	0
`count`	0

$\varphi_{m(Fib)}$:

`fibonacci`	$\langle$ `int, n,`	`{this.count = this.count + 1;`
		`if (n == 0) this.f = 0;`
		`else`
		`if (n == 1) this.f = 1;`
		`else this.f = fibonacci(n-1) +`
		`fibonacci(n-2);`
		`return this.f; }` $\rangle$

$$\sigma_1 = \langle\, \rho_0,\, \mu_0,\, \delta_1 \,\rangle$$

δ_1 :

`Fib`	$\langle\, \varphi_{v(Fib)},\, \varphi_{m(Fib)} \,\rangle$

Come sarà chiaro di seguito, la variabile di istanza `count` indicherà alla fine il numero di chiamate alla funzione `fibonacci` e NON la profondità del livello di annidamento delle chiamate come nell'esercizio in Sezione 7.4.

1.2) Binding del parametro del metodo `main` ed applicazione delle regole del sistema *local_decl* alle linee `28-29`

$$\sigma_2 = \langle\, \rho_1,\, \mu_1,\, \delta_1 \,\rangle$$

ρ_1 :

`foo`	*null*
`fib`	l_1
`res`	0

μ_1 :

l_1	$\langle\ \varphi_{v(l_1)}$: `f` 0 / `count` 0 , $\varphi_{m(Fib)} \,\rangle$

1.3) Applicazione di com_{assign} alla linea `31`: `res = fib.fibonacci(2);`

1.3.1) Prima applicazione di $exp_{mcallparnodecl}$ alla linea `31` sulla prima chiamata `fib.fibonacci(2)`.

$$\rho_1(\texttt{fib}) = l_1$$
$$\mu_1(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fib)} \,\rangle$$
$$\sigma_{local1} = \langle\, \rho_{local1},\, \mu_1,\, \delta_1 \,\rangle$$

ρ_{local1} :

n	2
this	l_1

μ_1 :

l_1	$\langle \ \varphi_{v(l_1)}$:	f	0	, $\varphi_{m(Fib)} \rangle$
		count	0	

1.3.2) Applicazione di com_{concat} alle linee `09-18`. Applicazione di com_{assign_this} alla linea `09`. Applicazione di com_{if_false} alle linee `12-18`, ancora applicazione di com_{if_false} alle linee `15-18` e quindi applicazione di com_{assign_this} alla linea `18`: `this.f = this.fibonacci(1) + this.fibonacci(0);` (ACTIVE).

$$\sigma'_{local1} = \langle \rho_{local1}, \mu_2, \delta_1 \rangle$$

ρ_{local1} :

n	2
this	l_1

μ_2 :

l_1	$\langle \ \varphi_{v(l_1)}$:	f	0	, $\varphi_{m(Fib)} \rangle$
		count	1	

Allo stesso modo dell'esercizio in Sezione 7.4, μ_2 dovrebbe rappresentare l'heap μ_1 dopo che `f` è stato modificato dall'assegnamento `this.f = this.fibonacci(1) + this.fibonacci(0);`. In questo caso abbiamo che nel lato destro dell'assegnamento ci sono due chiamate ricorsive alla funzione `fibonacci` e quindi l'assegnamento non può terminare ma deve attendere la terminazione di tali chiamate. Per questo motivo lo stack locale ρ_{local1} rimane attivo e l'heap μ_2 non riporta subito la modifica su `f` che per ora rimane a zero. Quindi, si applica di nuovo la regola $exp_{mcallparnodecl}$ alla chiamata ricorsiva più a sinistra `this.fibonacci(1)`. In memoria abbiamo quindi due stack ρ e ρ_{local1}, ma solo ρ_{local1} è attivo.

1.3.2.1) Seconda applicazione di $exp_{mcallparnodecl}$ alla linea `18` su `this.fibonacci(n-1)` e quindi su `this.fibonacci(1)`.

$$\rho_{local1}(\texttt{this}) = l_1$$
$$\mu_2(l_1) = \langle \varphi_{v(l_1)}, \varphi_{m(Fib)} \rangle$$
$$\sigma_{local1.1} = \langle \rho_{local1.1}, \mu_2, \delta_1 \rangle$$

$\rho_{local1.1}$:

`n`	1
`this`	l_1

μ_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 0 `count` | 1	, $\varphi_{m(Fib)}$ $\rangle$

1.3.2.2) Applicazione di com_{concat} alle linee `09-18`. Applicazione di com_{assign_this} alla linea `09`. Applicazione di com_{if_false} alle linee `12-18`, applicazione di com_{if_true} alle linee `15-18` e quindi applicazione di com_{assign_this} alla linea `16`: `this.f = 1;`

Avendo scelto il ramo *true* dell'`if` più interno, l'assegnamento `this.f = 1;` alla linea `16` può essere completato con la regola com_{assign_this}. Si può quindi passare alla linea `21` ad eseguire il `return` finale della chiamata ricorsiva `this.fibonacci(1)` più a sinistra di `this.f = this.fibonacci(1) + this.fibonacci(0);`. Quindi, $\sigma'_{local1.1}$ di seguito riportato è lo stato locale a livello di profondità 2 subito dopo la valutazione dello statement `return this.f;` (cioè `return 1;`).

$$\sigma'_{local1.1} = \langle\, \rho_{local1.1},\, \mu_3,\, \delta_1 \,\rangle$$

$\rho_{local1.1}$:

`n`	1
`this`	l_1

μ_3 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 1 `count` | 2	, $\varphi_{m(Fib)}$ $\rangle$

Dopo questo `return`, essendosi conclusa la chiamata a `this.fibonacci(1);` del punto 1.3.2.1, si può risalire al livello 1. Però, a tale livello, l'assegnamento `this.f = this.fibonacci(1) + this.fibonacci(0);` (che ora è diventato `this.f = 1 + this.fibonacci(0);`) nel punto 1.3.2 non può ancora essere completato in quanto bisogna ancora valutare `this.fibonacci(0)`.

Comunque, da questo momento in poi, lo stack locale $\rho'_{local1.1}$ in $\sigma_{local1.1}$ non serve più e può essere cancellato dalla memoria[15]. Ritorna invece attivo lo stack ρ_{local1} (creato con la prima chiamata a `fib.fibonacci(2)` nei punti 1.3.1 e 1.3.2) e quindi lo stato corrente è il seguente:

[15]Come già osservato nella Sezione 7.4, questa operazione di cancellazione in Java è effettuata dal *garbage collector* non preso in considerazione in queste note.

$$\sigma''_{local1} = \langle\, \rho_{local1},\, \mu'_2,\, \delta_1 \,\rangle$$

ρ_{local1} :

`n`	2
`this`	l_1

μ'_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 1 `count` | 2	, $\varphi_{m(Fib)}$ $\rangle$

Si nota che, a livello 1, la variabile `n` nello stack attualmente attivo ρ_{local1} vale ancora 2. Infatti, la chiamata di `this.fibonacci(n-2)` si attualizza con `this.fibonacci(0)`.

Inoltre, l'heap μ'_2 è uguale all'heap μ_3. Come già fatto nella Sezione 7.4, si è preferito usare il pedice "$_2$" in μ'_2 per indicare che si risaliti ad un livello superiore.

1.3.2.3) Terza applicazione di $exp_{mcallparnodecl}$ alla linea `18` su `this.fibonacci(n-2)` e quindi su `this.fibonacci(0)`.

$$\rho_{local1}(\texttt{this}) = l_1$$
$$\mu'_2(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fib)} \,\rangle$$
$$\sigma_{local1.2} = \langle\, \rho_{local1.2},\, \mu'_2,\, \delta_1 \,\rangle$$

$\rho_{local1.2}$:

`n`	0
`this`	l_1

μ'_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 1 `count` | 2	, $\varphi_{m(Fib)}$ $\rangle$

1.3.2.4) Applicazione di com_{concat} alle linee `09-18`. Applicazione di com_{assign_this} alla linea `09`. Applicazione di com_{if_true} alle linee `12-18` e quindi applicazione di com_{assign_this} alla linea `13`: `this.f = 0;`

Avendo scelto il ramo *true* dell'`if` più esterno, l'assegnamento `this.f = 0;` alla linea `13` può essere completato con la regola com_{assign_this}. Si può quindi passare alla linea `21` ad eseguire il `return` della chiamata ricorsiva `this.fibonacci(0)` più a destra di `this.f = this.fibonacci(1) + this.fibonacci(0);`. Quindi, $\sigma'_{local1.2}$ di seguito riportato è lo stato locale

a livello di profondità 2 dopo la valutazione dello statement `return this.f` (cioè `return 0`).

$$\sigma'_{local1.2} = \langle\, \rho_{local1.2},\, \mu'_3,\, \delta_1 \,\rangle$$

$\rho_{local1.2}$:

`n`	0
`this`	l_1

μ'_3 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 0 `count` | 3	, $\varphi_{m(Fib)}$ $\rangle$

Dopo questo `return`, essendosi conclusa la chiamata a `this.fibonacci(0);` del punto 1.3.2.3, si può risalire di nuovo al livello 1 e lo stack locale $\rho_{local1.2}$ in $\sigma'_{local1.2}$ non serve più e può essere cancellato dalla memoria. Ritorna invece attivo ancora una volta lo stack ρ_{local1}. Quindi, l'assegnamento `this.f = this.fibonacci(1) + this.fibonacci(0);` (che ora è diventato `this.f = 1 + 0;`) nel punto 1.3.2 può essere completato e si può quindi passare per l'ultima volta alla linea `11` ad eseguire il `return` finale della prima chiamata a `this.fibonacci(2);` del punto 1.3.1. In questo momento lo stato è il seguente:

$$\sigma'''_{local1} = \langle\, \rho_{local1},\, \mu''_2,\, \delta_1 \,\rangle$$

ρ_{local1} :

`n`	2
`this`	l_1

μ''_2 :

l_1	$\langle$	$\varphi_{v(l_1)}$:	`f` | 1 `count` | 3	, $\varphi_{m(Fib)}$ $\rangle$

Essendosi conclusa la prima chiamata a `this.fibonacci(2)` del punto 1.3.1, si può risalire al livello 0. A tale livello, l'assegnamento `res = fib.fibonacci(2);` (e cioè `res = 1;`) nel punto 1.3 può ora essere effettuato e lo stato finale subito prima di concludere la funzione `main` è:

$$\sigma_3 = \langle\, \rho'_1,\, \mu'_1,\, \delta_1 \,\rangle$$

ρ'_1 :

`foo`	*null*
`fib`	l_1
`res`	1

μ'_1 :

l_1	$\langle \ \varphi_{v(l_1)}$:	`f`	1
		`count`	3
	$, \varphi_{m(Fib)} \rangle$		

Può essere utile osservare che per gli stack locali ρ_{local1}, $\rho_{local1.1}$ e $\rho_{local1.2}$ si sono usati i pedici "$_1$", "$_{1.1}$" e "$_{1.2}$", rispettivamente. Questa indicizzazione sta semplicemente ad indicare che dal livello 1, in cui era attivo lo stack ρ_{local1} per la prima chiamata `this.fibonacci(2)` (vedi punto 1.3.2), si è richiamata ricorsivamente per la prima volta la funzione `fibonacci(n-1)` (alla linea `18`) attivando lo stack $\rho_{local1.1}$ in un livello inferiore. Poi, dopo essere risaliti al livello 1 e dopo aver riattivato lo stack ρ_{local1} (vedi punto 1.3.2.2), si è richiamata ricorsivamente per la seconda volta la funzione `fibonacci(n-2)` (sempre alla linea `18` all'interno della chiamata `this.fibonacci(2)`) attivando lo stack $\rho_{local1.2}$ (tornando, per la seconda volta, al livello inferiore 2). Per evidenziare le "discese" e le "risalite" delle chiamate (rispetto agli indici usati per lo stack) si può scrivere la sequenza in questo modo: $\rho_0 \longrightarrow \rho_1 \searrow \rho_{local1} \searrow \rho_{local1.1} \nearrow \rho_{local1} \searrow \rho_{local1.2} \nearrow \rho_{local1} \nearrow \rho'_1$.

Inoltre, come già osservato nella Sezione 7.4, deve sempre essere chiaro che la sequenza μ_0, μ_1, μ_2, μ_3, μ'_2, μ'_3, μ''_2, μ'_1 indica modifiche successive sempre allo stesso heap ed in particolare sempre alla stessa istanza `fib` riferita nell'heap da l_1. Per evidenziare le discese e le risalite delle chiamate (rispetto agli indici usati per l'heap) si può scrivere la sequenza in questo modo: $\mu_0 \longrightarrow \mu_1 \searrow \mu_2 \searrow \mu_3 \nearrow \mu'_2 \searrow \mu'_3 \nearrow \mu''_2 \nearrow \mu'_1$.

Vediamo ora la derivazione di valutazione della prima chiamata alla funzione `fibonacci` nello statement di riga 31 dallo stato $\sigma_2 = \langle \rho_1, \mu_1, \delta_1 \rangle$:

$$\frac{\begin{array}{c}\sigma_2 = \langle \rho_1, \mu_1, \delta_1 \rangle \\ \textbf{d1:} \langle \texttt{fib.fibonacci(2)}, \sigma_2 \rangle \longrightarrow_{exp_{mcallparnodecl}} \langle \underline{1}, \sigma'_2 \rangle \quad \sigma'_2 = \langle \rho_1, \mu''_2, \delta_1 \rangle \\ \rho'_1 = \rho_1[^{1}/_{\texttt{res}}] \quad \sigma_3 = \langle \rho'_1, \mu'_1, \delta_1 \rangle\end{array}}{\langle \texttt{res} = \texttt{fib.fibonacci(2);}, \sigma_2 \rangle \longrightarrow_{com} \sigma_3} \ (com_{assign})$$

Sotto-derivazione **d1:**

$$\frac{\begin{array}{c}\sigma_2 = \langle\, \rho_1,\, \mu_1,\, \delta_1 \,\rangle \quad \rho_1(\texttt{fib}) = l_1 \quad \mu_1(l_1) = \langle\, \varphi_{v(l_1)},\, \varphi_{m(Fib)} \,\rangle \\ \varphi_{m(Fib)}(\texttt{fibonacci}) = \langle\, \texttt{n},\, \texttt{MB}_{\texttt{Fib}} \,\rangle \quad \langle\, 2\,, \sigma_2 \,\rangle \longrightarrow_{exp_{const}} \langle\, \underline{2},\, \sigma_2 \,\rangle \\ \texttt{MB}_{\texttt{Fib}} = \{\texttt{S}_{\texttt{Fib}}\ \texttt{return this.f;}\} \\ \sigma_{local_1} = \langle\, \rho_{local_1},\, \mu_1,\, \delta_1 \,\rangle \quad \rho_{local_1} = \omega[\,{}^{\underline{2}}/_{\texttt{n}},\, {}^{l_1}/_{\texttt{this}}\,] \\ \langle\, \texttt{S}_{\texttt{Fib}}\,, \sigma_{local_1} \,\rangle \longrightarrow_{com_{concat}} \sigma'''_{local1} \\ \sigma'''_{local1} = \langle\, \rho_{local1},\, \mu''_2,\, \delta_1 \,\rangle \quad \langle\, \texttt{this.f}\,, \sigma'''_{local1} \,\rangle \longrightarrow_{exp_{ideobj}} \langle\, \underline{1},\, \sigma'''_{local1} \,\rangle \\ \sigma'_2 = \langle\, \rho_1,\, \mu''_2,\, \delta_1 \,\rangle \end{array}}{\langle\, \texttt{fib.fibonacci(2)}\,, \sigma_2 \,\rangle \longrightarrow_{exp} \langle\, \underline{1},\, \sigma'_2 \,\rangle} \quad (exp_{mcallparnodecl})$$

Dove $\texttt{S}_{\texttt{Fib}}$ rappresenta le linee di codice 09-18 e, come fatto precedentemente, μ''_2 (nello stato σ'_2 della regola com_{assign}) è stato semplicemente rinominato con μ'_1 (nello stato σ_3 della stessa regola) per indicare che, dopo la valutazione della chiamata a `fibonacci`, "si risale" ad un livello superiore (anche se μ''_2 è uguale a μ'_1).