PROGRAMMAZIONE II (A,B) - a.a. 2013-14

Seconda Valutazione Intermedia — 30 maggio 20144

Soluzioni proposte dai docenti

Esercizio 1. Si consideri il seguente programma a oggetti scritto in una sintassi Java-like:

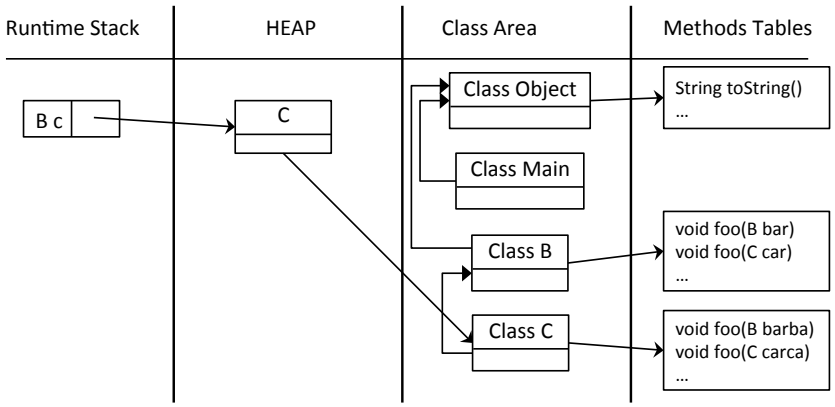
```
class B {
     public void foo(B bar) {
          System.out.print("B1 ");
     }
     public void foo(C car) {
          System.out.print("B2 ");
     }
}
class C extends B {
     public void foo(B barba) {
          System.out.print("C1 ");
     }
     public void foo(C carca) {
          System.out.print("C2 ");
     }
}
```

```
class Main {
     public static void main(String[] args) {
          (1) B c = new C();
          (2) B b = new B();
          (3) b.foo(c);
          (4) c.foo(b);
          (5) c.foo(c);
      }
}
```

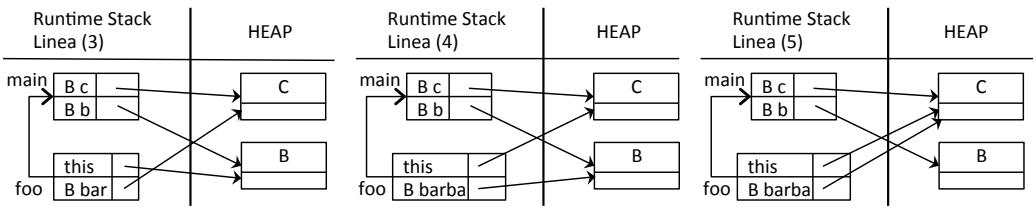
1. Cosa stampa il programma?

   Soluzione Stampa **"B1 C1 C1 "**. Infatti nelle linee (3), (4) e (5), il compilatore determina che il metodo da invocare è quello con firma `foo(B)`, perché in tutti i casi il parametro attuale ha tipo (statico) `B`. Poi a tempo di esecuzione, per il meccanismo di binding dinamico, in linea (3) viene eseguito il metodo `foo(B bar)` della classe `B`, mentre nelle linee (4) e (5) viene eseguito il metodo `foo(B barba)` della classe `C`.

2. Descrivere le informazioni presenti a run-time al termine dell'esecuzione del comando (1), e cioè il contenuto della class area, delle tabelle dei metodi, del run-time stack e dello heap.



3. Mostrare la struttura dei record di attivazione che vengono creati sullo stack in corrispondenza dell'invocazione del metodo `foo` nelle istruzioni (3), (4) e (5).



Esercizio 2. Per questo esercizio, si scelga uno solo dei due programmi seguenti, scritti rispettivamente in JavaScript e in OCaml ma sostanzialmente equivalenti. Indicare in modo chiaro la scelta fatta.

```
function dummy(x) {
    return 1000;
}
function evil(g,n) {
    function f(x) {
        return 10 + n;
    }
    if (n == 1) return f(0) + g(0); (*)
    else return evil(f,n-1);
}
evil(dummy,2);
```

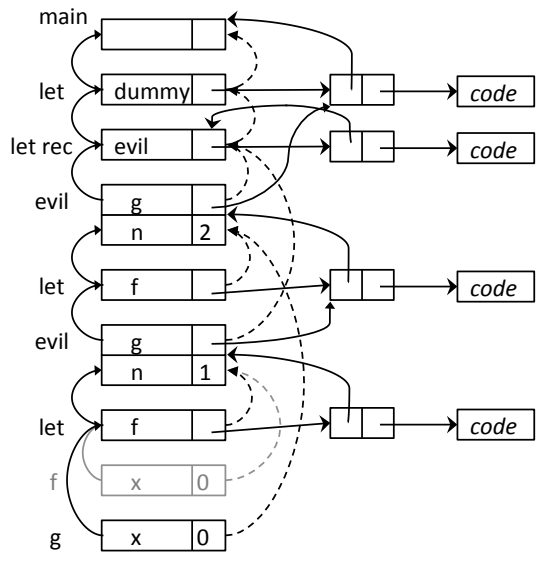
```
let dummy x = 1000 in
    let rec evil g n =
                (let f x = 10 + n in
                 if n = 1 then f 0 + g 0  (*)
                          else evil f (n - 1))
    in evil dummy 2
```

1. Determinare il tipo di tutti gli identificatori che compaiono nel programma scelto.
   Soluzione:

```
dummy : int -> int    (o anche     dummy : 'a -> int )
x : int               (o anche     x : 'a )
evil : (int -> int) -> int -> int
g : int -> int
n : int
f : int -> int        (o anche     f : 'a -> int )
x : int               (o anche     x : 'a )
```

2. Descrivere lo stato dello stack dei record di attivazione subito dopo l'invocazione della funzione g nella linea (*). Indicare per ogni record di attivazione le informazioni relative al puntatore di catena statica, puntatore di catena dinamica e struttura dell'ambiente locale.

Soluzione per codice OCaml:



Note:

(a) I puntatori di catena dinamica sono rappresentati a sinistra dei record di attivazione, con linee solide.

(b) I puntatori di catena statica sono rappresentati a destra dei record di attivazione, con linee tratteggiate.

(c) La chiusura di una funzione punta all'ambiente in cui la funzione è dichiarata se essa è ricorsiva (come `evil`), altrimenti punta all'ambiente immediatamente più esterno.

(d) Il primo record di attivazione (*main*) rappresenta il top level di OCaml, contenente tutte le informazioni di sistema.

(e) Il penultimo record di attivazione (chiamata di f, in grigio) viene distrutto prima della creazione dell'ultimo.

3. Qual è il valore calcolato dal programma?

Soluzione Il programma calcola 23. Infatti la valutazione dell'espressione $f(0) + g(0)$ nel corpo di *evil* invoca prima la seconda definizione di f, che restituisce 11 poiché n vale 1 nell'ambiente non locale, e poi la prima definizione di f, legata a g dal passaggio dei parametri, che restituisce 12 visto che n vale 2 nell'ambiente non locale.

Esercizio 3. Si consideri il linguaggio didattico imperativo senza funzioni né procedure. Estendiamo la sintassi dei comandi in modo da includere il comando condizionale multiplo **case-esac**, con la seguente sintassi concreta:

```
case
| E1 -->  CL1
| E2 -->  CL2
...
| En -->  CLn
esac
```

Nel comando, per ogni $i \in \{1, 2, \ldots, n\}$, `Ei` è una espressione booleana chiamata *guardia*, mentre `CLi` è una lista di comandi. Il numero n è un intero maggiore o uguale a zero.

L'esecuzione del comando case-esac consiste nel valutare sequenzialmente le guardie, e per ogni guardia che restituisce true nell'eseguire la corrispondente lista di comandi.

1. Estendere la sintassi astratta del linguaggio didattico imperativo in modo da includere il comando condizionale multiplo **case-esac**.
   Soluzione:

```
type com =
     | ...
     | Case of (exp * (com list)) list
```

2. Definire le regole di semantica operazionale della valutazione del comando case-esac e derivare il relativo codice OCaml per interpretare il comando. Si usino liberamente le funzioni OCaml:

```
semc: com * (dval env) * (mval store) -> (mval store)
semcl: (com list) * (dval env) * (mval store) -> (mval store)
sem:  exp * (dval env) * (mval store) -> eval
```

   Soluzione

   (a) Regole di semantica operazionale *big-step*:

$$\frac{}{amb, mem \triangleright \mathbf{Case}([\,]) \to mem}$$

$$\frac{amb, mem \triangleright guard \to true \quad amb, mem \triangleright coms \to mem_1 \quad amb, mem_1 \triangleright \mathbf{Case}(rest) \to mem_2}{amb, mem \triangleright \mathbf{Case}((guard, coms) :: rest)) \to mem_2}$$

$$\frac{amb, mem \triangleright guard \to false \qquad amb, mem \triangleright \mathbf{Case}(rest) \to mem_1}{amb, mem \triangleright \mathbf{Case}((guard, coms) :: rest)) \to mem_1}$$

   (b) Estensione della definizione della funzione `semc` per considerare il nuovo comando:

```
let rec semc (c: com) (amb: dval env) (mem: mval store) = match c with
     | ...
     | Case ([]) -> mem
     | Case ((guard, coms) :: rest) -> match sem(guard, amb, mem) with
               | Bool(true) -> semcl(coms@[Case(rest)], amb, mem)
               | Bool(false) -> semc(Case(rest), amb, mem)
               | _  -> failwith "Non boolean guard..."
```

Esercizio 4. La classe Java `Stack` nel box a sinistra implementa una semplice pila con i metodi `pop()` e `push()`, utilizzando un *array parzialmente riempito*. Uno studente osserva che il codice della `pop()` può essere semplificato, e realizza la sottoclasse `CleverStack` mostrata nel box a destra.

```
public class Stack{
    protected Object [] stack = new Object[1000];
    protected int size = 0;
    void push(Object obj){
        stack[size] = obj;
        size++;  }
    Object pop(){
        if (size == 0) return null;
        else {
            size--;
            Object tmp = stack[size];
            stack[size] = null;
            return tmp;        }  }  }
```

```
class CleverStack extends Stack{
    Object pop(){
        if (size == 0) return null;
        else {size--;
              return stack[size]};
    }
}
```

1. Confrontare in modo informale il comportamento delle due classi rispetto all'occupazione di memoria a run-time.

   Soluzione Il metodo `pop()` della classe `Stack` sovrascrive con `null` il riferimento nell'array `stack` all'oggetto restituito. Quindi se non vi sono altri riferimenti all'oggetto, questo diventa garbage e può essere reclamato dal garbage collector. Invece il metodo `pop()` di `CleverStack` non sovrascrive il riferimento all'oggetto restituito, quindi anche se non vi sono altri riferimenti all'oggetto esso rimane *live* (raggiungibile) anche se non è più accessibile logicamente, e il garbage collector non lo può reclamare. Quindi in generale un programma occuperà più memoria (nello heap) usando la classe `CleverStack` al posto di `Stack`.
   Nota: l'allocazione della variable `tmp` nel metodo `pop()` di `Stack` è trascurabile, poiché essa viene reclamata all'uscita del metodo.

2. Fornire un programma che usa un oggetto di tipo `Stack`, e tale che se lo si sostituisce con un oggetto di tipo `CleverStack` si ottiene un comportamento diverso.

Soluzione

Nota: Il codice di `Stack` nella versione originale del testo non conteneva i modificatori di visibilità `protected`, ma durante lo scritto è stato detto che il programma richiesto poteva usare solo i metodi `push()` e `pop()`. Si consideri il seguente programma:

```
    public static void main (String [] args){
        final int SIZE = 10000000; // SIZE * 64 <  (dimensione in bit dello heap)
        Stack s;                   // SIZE * 64 * 2 > (dimensione in bit dello heap)
        if (args[0].equals("Stack")) s = new Stack();
            else                     s = new CleverStack();
        s.push(null);
        s.push(new double[SIZE]);
        s.pop();
        s.pop();
        s.push(new double[SIZE]);
    }
```

Se eseguito con `java Stack Stack`, il programma usa `Stack` e l'esecuzione termina normalmente. Se eseguito con `java Stack QualunqueAltraStringa`, il programma usa `CleverStack` e abortisce con errore:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at Stack.main(Stack.java:26)
```

Infatti nel secondo caso l'oggetto allocato con la seconda `push()` non viene reclamato con la `pop()`, e l'allocazione di un secondo oggetto di uguale dimensione nell'ultima istruzione fallisce per l'esaurimento dello heap.