Correzione Primo Esonero

Ivano Salvo
Sapienza Università di Roma
email: salvo@di.uniroma1.it

Anno Accademico 2011-12

Esercizio 1.1

Gli output corretti sono:

```
/*p0*/ F: 2
/*p1*/ F: 5
/*p2*/ F:S:( 1, 1)
/*p3*/ F:S:( 2, 2)
/*p4*/ F:D: 2
/*p5*/ F:D: 5
/*p6*/ cannot find symbol
       symbol  : constructor S()
       location: class S
   S s1 = new S(); s1.print();
/*p7*/ F:D: 2
/*p8*/ F:D: 5
/*p9*/ incompatible types
       found   : S
       required: D
       D d2 = new S(2,2); d2.print();
```

Osservazioni ed Errori Tipici

L'errore più tipico stato dichiarare l'output `F:S:(3,1)` al punto `/*p3*/`. Questo errore deriva chiaramente dal non aver compreso che il metodo `n()` definito nella classe `F` viene ridefinito nella classe `S`. Siccome al momento della prima invocazione di `m` con `f2.m()`, `this` viene legato a `f2` e quando l'esecuzione del metodo `m` causa l'invocazione di `n()` viene invocata da `f2`, il cui tipo dinamico è `S`, il cui effetto è di raddoppiare la variabile di istanza `s` della classe `S` (a differenza del metodo `n` definito in `F` che raddoppia la variabile di istanza `f`).

Un buon numero di studenti (maggiore di quanto mi sarei potuto aspettare), ha ritenuto che i punti `p6`–`p9` causassero un errore perché il metodo `print` o il metodo `m()` non sono definiti sulle classi `D` od `S`. Ciò è falso, per via dell'ereditarietà.

Esercizio 1.2

C'erano ovviamente molte soluzioni possibili. Ecco un paio di molte gettonate:

```
public static int multiplo1(int m, int n){
    for (int k=0; ; k++) if (m==k*n) return k;
    }

public static int multiplo2(int m, int n){
    int k=0;
    while (m!=n*k) k++;
    return k;
    }
```

Io avrei preferito qualcosa del tipo:

```
public static int multiploProf(int m, int n){
    int s=0;
    int k=0;
    while (s!=m){
    /* INV: s=k*n */
         s+=n;
         k++;
    }
    return k;
}
```

Per quanto riguarda la versione ricorsiva, un approccio seguito da molti è stato quello di tradurre il ciclo `while` con chiamate ricorsive. Per fare ciò era necessario usare una funzione con un parametro ausiliario. IL risultato era qualcosa del tipo:

```
public static int multiploRec(int m, int n){
   return multiploRec(m,n,0);
   }

public static int multiploRec(int m, int n, int k){
   if (m==n*k) return k;
   else return multiploRec(m,n,k+1);
   }
```

Non era in realtà necessario usare un ulteriore parametro. Una interessante soluzione è la seguente:

```
public static int multiploProfRec(int m, int n){
   if (m==0) return 0;
   else return 1+multiploProfRec(m-n,n);
   }
 }
```

Osservazioni ed Errori Tipici

Un errore comune (che ho sostanzilamente perdonato), è quello di fare chiamate ricorsive del tipo `multiploRec(m,n,k++)` (questo vale anche per la soluzione ricorsiva all'esercizio 1.3). Attenzione che questa chiamata semplicemente non termina, perché la semantica dell'incremento postfisso `++` dice: "valuta l'espressione, poi incrementa". Quindi, le chiamate ricorsive entrano in loop, perchè **non** si modifica il terzo parametro. Funzionerebbe l'incremento *prefisso*, `++k`, ma mi permetto di osservare, che in questo caso non c'è nessun interesse a modificare il valore della variabile `k`, e ci interessa solo trasmettere alla successiva attivazione il valore `k+1`.

Esercizio 1.3

Chiaramente il metodo rovescia (o inverte) il contenuto di un qualsiasi vettore di interi. Si può produrre un errore run-time solo se il riferimento `a` è il puntatore NULL. Volendo essere precisi, la postcondizione si poteva scrivere come $\forall i : 0 \leq i < a.length - 1 . a_{\text{post}}[i] = a[a.length - i - 1]$.

Vediamo due possibili tracce di esecuzione. Era sufficiente stampare i valori `l`, `r` e del vettore `a` all'inizio del ciclo e alla fine di ogni iterazione. Ecco una traccia con 4 elementi:

```
l=0 r=3 a={ 3, 1, 4, 7}
l=1 r=2 a={ 7, 1, 4, 3}
l=2 r=1 a={ 7, 4, 1, 3}
```

Ed una con 5 elementi:

```
l=0 r=4 a={ 3, 1, 4, 7, 1}
l=1 r=3 a={ 1, 1, 4, 7, 3}
l=2 r=2 a={ 1, 7, 4, 1, 3}
```

Il metodo termina sempre, ed è facile dimostrare che la funzione $r - l$ soddisfa a tutti i requisiti per essere una funzione di terminazione. Infatti, la guardia del ciclo implica che $r > l$, e quindi $r - l > 0$ durante l'esecuzione del ciclo.

Siccome il suo valore decresce di 2 ad ogni iterazione, abbiamo anche che $\lfloor a.length/2 \rfloor$ è una facile stima del numero di iterazioni.

La più naturale versione ricorsiva è quella che si ottiene semplicemente traducendo in modo standard il ciclo `while`. Si usano due parametri ausiliari per rappresentare le variabili `l` ed `r` e ad ogni chiamata si incrementa la prima e si decrementa la seconda, esattamente come nel ciclo `while`. E come nel ciclo `while` ci si ferma quando `l` non è minore di `r`.

```
public static void m(int [] a, int l, int r){
    if (r<=l) return;
    int h=a[r];
    a[r]=a[l];
    a[l]=h;
    m(a,l+1,r-1);
}
```

Con uno sforzo di fantasia, osservando le simmetrie del problema, si poteva anche usare un unico parametro ausiliario, a scapito però della chiarezza del codice.

```
public static void m(int [] a, int j){
    if (j>=a.length/2) return;
    int h=a[a.length-j-1];
    a[a.length-j-1]=a[j];
    a[j]=h;
    m(a,j+1);
}
```

Osservazioni ed Errori Tipici

Un errore sconcertante è mettere precondizioni che dipendono dai valori di variabili come `l` ed `r` locali alle procedure. Le precondizioni dipendono dagli input di un metodo, quindi sostanzialmente *dal valore dei parametri* (o eventualmente da loro relazioni con lo stato dell'oggetto ricevente).

Precondizioni del tipo: "a vettore di interi" non sono scorrette, ma sono sussunte dai vincoli imposti dal sistema dei tipi. Non è chiaro cosa significhi "a vettore non vuoto", mentre è senza dubbio sbagliato porre precondizioni del tipo "$a.length > 1$ o $a.length > 2$, perchè in tali casi, il metodo funziona senza problemi (alcuni sono sembrati inorriditi dal fatto che non si entrasse mai nel ciclo). Chiuderei il discorso dicendo che una precondizione potrebbe essere stata viceversa "$a \neq$Null", ma usualmente questa viene data come implicita.

Più controversa una precondizione del tipo: "a vettore ordinato in modo decrescente". Questo metodo potrebbe essere usato in questo modo, per invertire l'ordine di un array. In tal caso, bisognava essere coerenti e dare delle postcondizioni coerenti (nel nostro caso: "a vettore ordinato in modo crescente").

Esercizio 2.1

Per fare questo esercizio è innanzitutto necessario capire che la funzione `scomponi` da' come risultato una coppia di fattori qualsiasi e noi non possiamo in nessun modo speculare su proprietà di questi fattori. Spero che la cosa non vi sorprenda: a volte una specifica è rilassata e noi non possiamo fare ipotesi sul fatto che una funzione dia una particolare soluzione canonica. Spesso ci sono buoni motivi per operare in questo modo. Nel nostro caso, potrebbe essere l'uso di una procedura che tenta di scomporre un numero facendo tentativi causali.

Fatta questa premessa, è chiaro che se n è primo, allora, lui è anche il suo massimo fattore primo. Viceversa il massimo fattore primo di un numero $n = fat_1 \cdot fat_2$ sarà il maggiore tra il massimo fattore primo di fat_1 e il massimo fattore primo di fat_2.

Abbiamo il nostro schema induttivo, con tanto di caso base. Prima di vedere il programma, vediamo la definizione del tipo `IntPair`. Osserviamo solo che si tratta sostanzialmente di una classe che definisce coppie di valori, che non necessita di particolari protezioni di visibilità, nè di definizione di operazioni.

```
class IntPair{
   public int fat1;
   public int fat2;
}
...

   public static int maxPrimo(int n){
       IntPair d;
       if (scomponi(n,d))
              return max(maxPrimo(d.fat1), maxPrimo(d.fat2));
         else return n;
   }
```

La versione iterativa, viceversa, è estremamente più complicata e richiede di memorizzare tutti i fattori trovati in una struttura dati (ad esempio un vettore di lunghezza $\log_2 n$, visto che un numero ha al più $\log_2 n$ fattori) e poi continuare a scomporli finchè non si arriva a fattori primi o fattori più piccoli del più grande primo già calcolato. Lasciamo tale implementazione all'esercizio del lettore.

Esercizio 2.2

La soluzione più semplice e naturale consiste nel rappresentare un intero come un naturale (per questo abbiamo il tipo `Nat`) e un booleano per rappresentare il segno. Lode a chi ha usato i tipi enumerati, però riflettete sul fatto che il tipo booleano altri non è che il tipo enumerato con 2 elementi (più ovviamente le usuali operazioni).

Detto questo, si può facilmente ricondurre le usuali operazioni sugli interi, ad operazioni sui naturali distinguendo alcuni casi che dipendono dei segni. Solo per

fare un esempio, per sommare due interi, devo prima guardare i segni. Se sono concordi, il segno del risultato è il segno degli operandi, e il loro modulo uguale alla somma dei moduli. Se sono discordi, occorrer sottrarre dall'intero con il modulo più grande il modulo più piccolo, e il segno del risultato sarà il segno del numero più grande in modulo. Di seguito il codice completo di una possibile soluzione.

Osservo che non conveniva definire gli interi come sottoclasse dei naturali.

```
class Int{
/* OVW: classe per rappresentare gli interi MUTABILI
 * un intero  rappresentato da un naturale e un booleano che
 * rappresenta il segno. true significa positivo e false negativo
 */
    boolean segno;
    Nat num;

    public Int(){
        num=new Nat();
        segno=true;
    }

    public Int(boolean s, Nat n){
        num=n;
        segno=s;
    }

    private Int opposto(){
       return new Int(!segno, num);
    }

    public void piu(Int z){
        if (segno==z.segno) num=num.piu(z.num);
        else if (num.minUg(z.num)) {
               num=z.num.meno(num);
               segno=z.segno;
               }
        else num=num.meno(z.num);
    }

    public void meno(Int z){
        piu(z.opposto());
    }

    public void per(Int z){
        segno= segno==z.segno;
        num=num.per(z.num);
    }
```

```
    public void div(Int z){
        segno= segno==z.segno;
        num=num.div(z.num);
    }

    public boolean minUg(Int z){
        if (segno && z.segno) return num.minUg(z.num);
        else if (segno) return false;
        else if (z.segno) return true;
        else return z.num.minUg(num);
    }

    public boolean equals(Object o){
        if (! (o instanceof Int)) return false;
          else return equals((Int) o);
    }

    public boolean equals(Int z){
        if (segno!=z.segno) return false;
          else return num.equals(z.num);
    }

    public String toString(){
        if (segno) return num.toString();
          else return "-"+num.toString();
    }
} // fine class Int
```