Completamenti, Higher Order e Riuso:

Sommario: 29 Maggio, 2018

- **Completamenti: Riuso**
  - > Linguaggi Procedurali
  - > Linguaggi Funzionali
  - > Linguaggi Object Oriented
- **Tipo Astratto `Queue` Immutable in OCaml**
  - > API per un Tipo Astratto Queue Immutable in OCaml
  - > Implementazione: La `coda funzionale`
  - > Stato Concreto in OCaml: AF & I
  - > Le operazioni già definite: costruttore mk e e presentazione toString
  - > Le Operazioni Osservatori: isEmpty, isIn, get
  - > Le Operazioni Produttori: add, remove, removeAll
- **Estensione e Riuso di Tipo Astratto `Queue` in Java**
  - > Contesto: API di `Queue`
  - > La nuova API di `QueueV1`
  - > Stato Concreto: Estensione, AF & I
  - > Costruttori: Estensione
  - > Eccezioni e Operazioni: Riuso e Overriding
  - > Gli Additional: `equals`, `toString`, `clone`
- **Esercizi**

Riuso di Codice nei Linguaggi di Programmazione

Meccanismi per Riuso di Codice nei Linguaggi di Programmazione:

- **Procedurali**:
  - Astrazioni di Controllo:
    - Pocedure e Funzioni
    - Trasmissione di Parametri: Valore, Riferimento, Nome,...
  - Astrazioni di Dati:
    - Definizione di Nuovi Tipi di Dato: `type`, `typedef`,...
    - Espressioni di Tipo (anche ricorsive): `Struct`, `record`,...
    - Tipi Astratti: API, ADT, Moduli
    - Librerie: Moduli
- **Funzionali**:
  - Astrazioni di Controllo: (v. procedurale ma senza stato)
  - Astrazioni di Dati: (v. procedurale) + Polimorfismo Generico
  - Higher Order:
    - Funzioni come Valori Principali (first class)
    - Combinatori: fold_left, fold_right, map, filter, ...

Riuso di Codice nei Linguaggi di Programmazione /2

Meccanismi per Riuso di Codice nei Linguaggi di Programmazione:

- **Object Oriented**:
  - Astrazioni di Controllo: (v. procedurale)
  - Astrazioni di Dati: (v. procedurale) + Polimorfismo Generico
  - Classe e Oggetti. Una forma di Higher Order
    - Un oggetto o è un "valore con le sue operazioni"
    - Trasmettere o è "trasmettere anche i metodi associati"
  - Gerarchie di Classi.
    - Ereditarietà
    - Overriding: Definito con riuso del super.metodo
    - Polimorfismo Object, di Sottotipo e Generico

Tipo Astratto Queue Immutable in OCaml

Un Problema:

Un Tipo Astratto 't queue per code polimorfe di valori di tipo generico 't e' un valore strutturato IMMUTABLE dotate delle seguenti operazioni con signature funzionale:

isEmpty: 't queue -> bool

isIn: 't queue -> 't -> int *(quante occorrenze del secondo argomento)*

get: 't queue -> 't

add: 't queue -> 't -> 't queue

remove: 't queue -> 't queue

removeAll: 't queue -> 't -> 't queue *(rimuove gli uguali al secondo argomento)*

e, in aggiunta:

+ un costruttore pubblico che inizializza la coda agli elementi della lista a cui e' applicato;

+ eccezioni pubbliche che si ritenga utile aggiungere;

+ toString 't queue -> string

Si forniscano/completino API ed ADT con AF ed I e si esegua il test nel file accluso

Un Contesto di Uso del Tipo Astratto da definire

Completare API e ADT sotto, e si esegua il test nel file accluso

```
module type QUEUE =
  sig
    type 't queue
    ...
    val toString: 't queue -> ('t -> string) -> string
  end;;

module Queue =
(struct
 type 't queue = Q of 't list * 't list
 (*
  AF(c) =
   I(c) =
 *)
 let mk xx = Q([],xx)
   ...
 let toString (Q(inp,out)) toStringT =
       let c = out @ (List.rev inp) in
       "<"^(List.fold_right (fun x a -> (toStringT x)^", "^a) c "<")
 end:QUEUE);;
```

Higher Order: Dove, Perchè, Quale Codice è in Riuso?

API per Tipo Astratto Queue Immutable

- Da completare con Eccezioni contestualmente alla definizione di un ADT

```
module type QUEUE =
  sig
    type 't queue
    exception InvalidOperation of string * string
    val mk: 't list -> 't queue
    val isEmpty: 't queue -> bool
    val isIn: 't queue -> 't -> int
    val get: 't queue -> 't
    val add: 't queue -> 't -> 't queue
    val remove: 't queue -> 't queue
    val removeAll: 't queue -> 't -> 't queue
    val toString: 't queue -> ('t -> string) -> string
  end;;
```

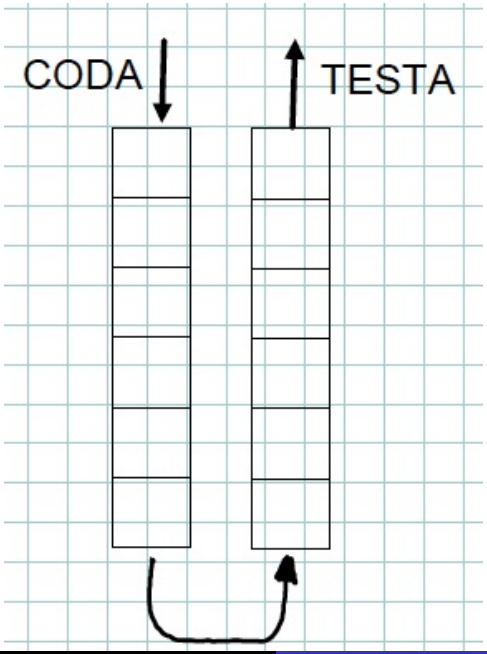
> La forma <v1,...,vn< della presentazione di una coda si desume dalla definizione di `toString()`, data nell'ADT da completare.

Implementazione: La coda funzionale

- **Lista funzionale** si comporta, attraverso le sue operazioni `::`, `hd`, `tl`, come una "pila" (first-in/last-out).
- **Usare** una lista funzionale per emulare una "coda" (last-in/last-out) richiede $O(n^2)$ accessi per aggiungere `n` elementi in coda.

Coda funzionale utilizza 2 liste accoppiate `(inp,out)`

- `inp` per aggiungere in testa al costo $O(1)$ accessi;
- `out!=[ ]` per rimuovere dalla testa al costo $O(1)$ accessi;
- `out == [] & inp!=[ ]`. La coda diventa; `(inp',out')` al costo $O(n)$ accessi, dove: `inp' == [ ]` e `out' == (rev inp)`;
- `n` `add/get/remove` richiedono $O(n)$ accessi;



Stato Concreto in OCaml: AF&I

- **Stato Concreto:**AF&I

```
module Queue =
(struct
 type 't queue = Q of 't list * 't list
 (*
 AF(c) = □ iff c == Q(inp,out) && length(in @ out)==0
 AF(c) = [x1,...,xn,y1,...ym] iff c == Q(inp,out) && length(inp @ out)>0
                                  out == [x1,...,xn] && (rev inp == [y1,...,ym]
 I(c) = true
 *)
```

> `I(c) = true` perchè ogni coppia di lista di uno stesso generico `'t` può correttamente essere interpretata, in accordo a `AF(c)`, come una coda

Le operazioni gi'a definite: `mk`, `toString`

- `mk: 't list -> 't queue.`

```
let mk xx = Q(□,xx)
```

  > Costruisce una coda pronta per potervi inserire ed estrarre oggetti in tempo `O(1)`.

- `toString: 't queue -> ('t -> string) -> string.`

```
let toString (Q(inp,out)) toStringT =
      let c = out @ (List.rev inp) in
      "<"^(List.fold_right (fun x a -> (toStringT x)^", "^a) c "<")
```

  > Higher Order per utilizzare presentazione,`toStringT`, specifiche per il tipo di oggetti in coda.
  > Usiamo `List.fold_right` più adatto per l'iterazione realizzata.
  > Osservare la funzione `fun x a -> (toStringT x)^", "^a`, iterata per costruire la stringa attesa..

Le Operazioni Osservatori: isEmpty, isIn, get

- `isEmpty: 't queue -> bool.`

```
let isEmpty (Q(inp,out)) = (List.length inp)+(List.length out) == 0
```

  > Controlla il numero di oggetti nelle due liste.

- `isIn: 't queue -> 't -> int .`

```
let isIn (Q(inp,out)) x = List.length (List.filter((==)x)(inp@out))
```

  > Higher Order: `List.filter((==)x)` riduce la lista ai soli uguali ad `x`, funzione `((==)x)`.

- `get: 't queue -> 't.`

```
let get (Q(inp,out)) =
    match (inp,out) with
            |([],[]) -> raise (InvalidOperation("get","queue is empty"))
            |(_,x::xs) -> x
            |_ -> List.hd(List.rev inp)
```

  > L'ultimo caso dovrebbe essere evitato per quanto possibile (v. remove).

Le Operazioni Produttori: add, remove, removeAll

- `add: 't queue -> 't -> 't queue.`

```
let add (Q(inp,out)) y = Q(y::inp,out)
```

  > Aggiunge un oggetto in tempo `O(1)` .

- `remove: 't queue -> 't queue.`

```
let remove (Q(inp,out)) =
   match (inp,out) with
           |(□,□) -> raise (InvalidOperation("Remove","queue is empty"))
           |(_,x::□) -> Q(□,List.rev inp)
           |(_,x::xs) when xs<>□ -> Q(inp,xs)
           |_ -> Q(□,List.tl(List.rev inp))
```

  > Il secondo caso anticipa la reverse di `inp`, e la sua ricollocazione come `out`, riducendo le situazione ricordate in `get`.

- `removeAll: 't queue -> 't -> 't queue.`

```
let removeAll (Q(inp,out)) z =
      let p = fun x -> x<>z in Q(List.filter p inp, List.filter p out)
```

  > Higher Order: Ancora `List.filter` ma per lasciare in ciascuna lista solo gli oggetti diversi da `z`, mediante la funzione `fun x -> x<>z`.

Estensione e Riuso di Tipo Astratto Queue in Java

- Il Problema.

Un Tipo Astratto 't queue per code polimorfe di valori di tipo generico 't \`e un valore strutturato MUTABLE dotato delle operazioni che possiamo vedere nella API, API<T>, allegata.
API<T> non contiene operazioni sulla dimensione corrente e ancor meno, sulla dimensione massima che ha avuta la coda nella sua storia passata.
Si chiede di estendere il tipo astratto in un nuovo tipo che contenga anche le operazioni size e maxsize. Allo scopo, si completino i file allegati:
+ APIV1.java
+ QueueV1.java
+ mewExcV1.java
+ TestsV1.java

Prologo: Intero sviluppo in package di nome QueuePack
Prologo: Usare i 4 file dati
(a) Fornire API
(b) Fornire Stato Concreto, AF e C
(c-d-e-f) Sviluppo:
(c) Fornire successiva operazione nell'ordine e
ad ogni definizione procedere con (d),(e),(f)
(d) Completare la segnatura con le eccezioni sollevate
(e) Aggiungere definizione classi di eccezioni relative
(f) compilare ed eseguire test di uso

Contesto: API di Queue

- Dobbiamo estendere un Tipo Astratta avente interfaccia:

```
public interface API<A> extends Cloneable{
        public boolean isEmpty();
    public int isIn(A x);
    public A get() throws EmptyQueueException;
    public void add(A x);
    public void remove() throws EmptyQueueException;
    public void removeAll(A x);
}
```

- Occore esaminare l'interfaccia per:
  - > Gli additionals da considerare obbligatoriamente: vedi "extends Cloneable"
  - > Operazioni e segnatura esatta ereditata
  - > Considerare le operazioni che devono essere overridden e la segnatura da usare
  - > Le operazioni su cui possiamo contare nella definizione delle nuove operazioni.

La nuova API di QueueV1

- La nuova APIV1<A>:

```
public interface APIV1<A> extends API<A>,Cloneable{
    public int size();
    public int maxsize();
}
```

- Come è fatta:
  - > È sottoclasse di API<A>, quindi:
  - > Ogni sua implementazione, ADT<A>, definisce tutte le operazioni di API<A>
  - > In aggiunta ha le 2 operazioni attese,
  - > con le eccezioni che eventualmente aggiungeremo alla segnatura durante lo studio delle loro definizioni.
  - > L'indicazione di "Cloneable" poteva essere omessa perchè....

Contesto di Uso fornito per QueueV1<A>

- Completare il seguente file sorgente:

```
package QueuePack;
import java.io.*;
import java.util.*;

public class QueueV1<A> extends Queue<A> implements APIV1<A>{
    private int size;
    private int maxsize;
    /*
     AF(c) = ....
     I(c) = ...
     */
    public QueueV1(Vector<A> In){
        super(In);
        maxsize = size = In.size();
    }
    //public boolean isEmpty() ereditato?
    // public int isIn(A x) ereditato?
    // public A get() eredidato?
    // public void add(A x)
    // public void remove() eredidato?
    // public void removeAll(A x) eredidato?
    public int size(){
        ...
    }
    public int maxsize(){
        ...
    }
    // public String toString() ereditata?
    // boolean equals(Object x) ereditata?
    // public QueueV1<A> clone() throws CloneNotSupportedException{ereditata?
}
```

- File da cui partire e da costruire quando non ci viene fornito:
  > Qui contiene già la definizione dello Stato Concreto
  > Evidenzia in verde quanto deve essere considerato per fornire QueueV1<A>

QueueV1<A>: Stato Concreto, AF&I

- Lo **Stato Concreto** quì è banale:
  > È quello ovvio con i soli field necessari per le nuove informazioni
  > A volte l'estensione richiede uno Stato Concreto che può duplicare le "strutture" preposte a contenere la stessa informazione nel vecchio (e inaccessibile stato) e nel nuovo stato.
  > Come sarebbe stata la situazione se avessimo richiesto di aggiungere il metodo `waitUp(x)` indicante quanti oggetti `x` ha davanti? (vedi esercizio...) .

```
public class QueueV1<A> extends Queue<A> implements APIV1<A>{
    private int size;
    private int maxsize;
    /*
     AF(c) = AF(c.super)
     I(c) = (c.maxsize >= c.size) &&
            (AF(c) == <x1,...,xn< <=> c.size == n, for n>=0)
     */
```

- Anche **AF** quì è banale, mentre **I**:
  > impone condizioni necessarie su `size`, `maxsize` e il valore espresso dall'oggetto. Ma
  > non garantisce che `maxsize` possa solo aumentare con la vita dell'oggetto

Costruttori: Estensione

```
public class QueueV1<A> extends Queue<A> implements APIV1<A>{
    private int size;
    private int maxsize;
    /*
     AF(c) = AF(c.super)
     I(c) = (c.maxsize >= c.size) &&
            (AF(c) == <x1,...,xn< <=> c.size == n, for n>=0)
     */
    public QueueV1(Vector<A> In){
        super(In);
        maxsize = size = In.size();
    }
```

- **Corpo Costruttore** ha questa tipica struttura:
  - > Se usa il super costruttore, l'invocazione di `super` è la prima azione;
  - > Quì è necessario invocare `super(In)` per costruire in modo appropriato il super oggetto;
  - > Il rimanente codice provvede ad inizializzare in nuovo Staito Concreto come richiesto da AF&I;

Riuso e Overriding in `isEmpty`, `isIn`, `get`, `add`, `remove`

```
public boolean isEmpty(){
    return (size == 0);
}
// public int isIn(A x) ereditato
// public A get() eredidato
public void add(A x){
    super.add(x);
    size++;
    if (size > maxsize) maxsize = size;
}
public void remove() throws EmptyQueueException{
    if (size == 0) throw new EmptyQueueException("remove");
    super.remove();
    size--;
}
```

- **Osservatori** La rappresentazione del valore non è cambiata
  > Potrebbero essere tutti ereditati, inclusa `isEmpty`, quì overridden;
- **Riuso e Overriding** di un metodo ha la struttura dell'overriding `add`
  > Invocazione di `super.add` con **riuso** del codice scritto per tale super metodo,
  > e Azioni specifiche che interessano il nuovo Stato Concreto: `size++; ... ;`
- **Remove** avremmo potuto omettere il controllo `size == 0` perchè...

Riuso e Overriding in `removeAll, size, maxsize`

```
public void removeAll(A x) {
    if ((size == 0)||(x == null)) return;
    int k = isIn(x);
    if (k == 0) return;
    super.removeAll(x);
    size = size - k;
}
public int size(){
    return size;
}
public int maxsize(){
    return maxsize;
}
```

- **RemoveAll** Anche quì anticipiamo `super.removeAll` e potremmo rimuovere il primo stm.
- **size e maxsize** sono osservatori ma operano sul nuovo Stato Concreto

Gli Additionals

```
// public String toString() ereditata
// boolean equals(Object x) ereditata
public QueueV1<A> clone() throws CloneNotSupportedException{
        QueueV1<A> ret = (QueueV1<A>) super.clone();
    ret.size = size;
    ret.maxsize = maxsize;
    return ret;
}
```

- **toString, Equals** Sono ereditate perchè operano solo sul vecchio Stato Concreto
  > Lo stesso per `elements` se fosse stata definita nella super
- **clone** Quì è molto diverso perchè `clone` deve operare sia sul vecchio Stato Concreto sia sul nuovo.
  > Questo è realizzato nell'usuale modo dell'overriding con riuso del codice di `super.clone`

Il Test

```
class Test1{//Corregere (1 errore usando il compilatore) ed eseguire
    public static void main(String[] args) throws EmptyQueueException,
        CloneNotSupportedException{
        Queue<String> q1;
        Vector<String> inlist = new Vector<String>();
        inlist.add("luigi");
        inlist.add("luca");
        inlist.add("laura");
        inlist.add("michele");
        q1 = new Queue<String>(inlist);
        System.out.println("the current status of q1 is: " + q1);
        QueueV1<String> q2 = new QueueV1<String>(inlist);
        System.out.println("the size of q2 is: " + q2.size());
        q2.remove();
        q2.add("luca");
        q2.add("giacomo");
        q2.add("luca");
        q2.add("luciano");
        q2.add("franca");
        q2.add("silvia");
        System.out.println("the current status of q2 is: " + q2);
        System.out.println("the current maxsize of q2 is: " + q2.maxsize());
        q2.removeAll("luca");
        System.out.println("the current status of q2 is: " + q2);
        q2.remove();
        q2.remove();
        q2.remove();
        System.out.println("the current status of q2 is: " + q2);
        System.out.println("the size of q2 is: " + q2.size() + ";  whilst its maxsize has been:
            " + q2.maxsize());
        QueueV1<String> q3 = q2.clone();
        q3.remove();
        System.out.println("the size of q3 is: " + q3.size() + ";  whilst its maxsize has been:
            " + q3.maxsize());
        System.out.println("the current status of q3 is: " + q3);
    }
}
```

Il Test: Compilazione e running

```
etruria-wifi-217-47:Mutable18 marcob$ javac APIV1.java newExcV1.java QueueV1.java TestsV1.java -d .
etruria-wifi-217-47:Mutable18 marcob$ java QueuePack/Test1
the current status of q1 is: <luigi, luca, laura, michele<
the size of q2 is: 4
the current status of q2 is: <luca, laura, michele, luca, giacomo, luca, luciano, franca, silvia<
the current maxsize of q2 is: 9
the current status of q2 is: <laura, michele, giacomo, luciano, franca, silvia<
the current status of q2 is: <luciano, franca, silvia<
the size of q2 is: 3;  whilst its maxsize has been: 9
the size of q3 is: 2;  whilst its maxsize has been: 9
the current status of q3 is: <franca, silvia<
etruria-wifi-217-47:Mutable18 marcob$
```