Linguaggi di Programmazione

Ivano Salvo

OOP4: Templates in C++
FP5: valutazione lazy

Corso di Laurea in **Matematica**

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **6**, 3 novembre 2020

Lezione 6a:

Templates
(aka: contaminazioni)

"Programmazione funzionale" in C++

Abbiamo visto come in Haskell si possa con grande naturalezza:

- definire **funzioni polimorfe**, che sono indipendenti dal tipo di dato contenuto in una struttura dati
- Si possa facilmente **operare su un'intera struttura dati** usando funzionali come **`map`** e **`fold`**.

Vediamo come questi concetti siano stati "interiorizzati" in C++ e più in generale nella programmazione a Oggetti o imperativa.

Il polimorfismo è stato introdotto in C++ sotto forma di **Templates** (in **Java** si chiamano **Generics**), mentre i funzionali in quelli che vengono chiamati **Algorithms** (noi vedremo per ora l'equivalente del map, ossia gli **iteratori**)

Templates

Un **template** altro non è che una struttura dati che **dipende da una variabile di tipo**. **Esempio**: definizione di uno Stack.

Si usa la notazione `<T>` per dire che lo Stack dipende da un tipo generico `T`.

Osservare come `T` sia usata come un qualsiasi tipo.

```
template <T>
class Stack<T>{
   int size; /* dimensione max dello stack */
   int t;  /* indice del top dello stack
              * indice primo posto libero */
   T* s; /* array per memorizzare gli elementi */
public:
   Stack();
   Stack(int);
   void push(T x);
   T pop();
   T top();
   /* altri metodi (dopo) */
}
```

Un po' di codice: costruttori

Notare le notazioni direi, "un po' pesanti".

Nel caso in cui non venga fornita in input al costruttore la dimensione dell'array, utilizzeremo un numero prefissato da una costante `MAX_SIZE` definita in una `#define`.

```
template<class T>Stack<T>::Stack(int maxSize){
   s = (T*) calloc(maxSize, sizeof(T));
   size = maxSize;
   t=0;
/* INV: t punta al primo elemento libero dell'array */
}

template<class T>Stack<T>::Stack(){
   s = (T*) calloc(MAX_SIZE, sizeof(T));
   size = MAX_SIZE;
   t=0;
/* INV: t punta al primo elemento libero dell'array */
}
```

Un po' di codice: metodi

```
template<class T>
bool Stack<T>::isEmpty(){
   return t==0;
}
template<class T>
bool Stack<T>::isFull(){
   return t>=size;
}
template<class T>
void Stack<T>::push(T x){
   assert(t<size); /* l'utente deve sapere! */
   s[t++]==x;
}
template<class T>
void Stack<T>::pop(T x){return s[--t];}

template<class T>
T Stack<T>::top(T x){return s[t-1];}
```

Esempio di Utilizzo

Quando si usa un template, **occorre specificare il tipo degli elementi che andranno inseriti**. Il compilatore C++ **genera una classe diversa per ogni tipo usato**, come si trattasse di una macro-espansione.

Il modo corretto di usare push sarebbe quello di chiedere preventivamente se lo stack è pieno…

```
int main(){
   Stack<Casella*> s();
   /* carico sullo stack le caselle di una
    * scacchiera */
   for (int i=0; i<NRIG; i++)
      for (int j=0; j<NCOL; j++)
         if (!s.isFull())
            s.push(new Casella(i,j));
   ...
}
```

Lezione 6a:

Valutazione Lazy

Riscaldamento: foldr e le sue sorelle

La funzione `myFold` in Haskell è definita con il nome di **`foldr`** (= **fold right**): è lo schema ricorsivo tipico di quando si raccolgono `al ritorno' i risultati di una ricorsione.

Ci sono dei casi in cui è utile `**spingere in avanti**' (sui parametri) dei risultati intermedi della computazione. Un esempio (già discusso a Informatica Generale): **rovesciamento di una lista**.

Possiamo ovviamente generalizzare questo schema ricorsivo…

```
 -- reverse inefficiente, perché usa ++ (lineare)
myReverse   []   = []
myReverse (x:xs) = myReverse xs ++ [x]
 -- da cui:
myReverse = foldr (\x y->y ++ [x]) []

 -- La versione efficiente aggiunge in testa
 -- su una lista ausiliaria...
reversEff xs = reversAux xs [] where
   reversAux   []   rs = rs
   reversAux (x:xs) rs = reversAux xs (x:rs)
```

Riscaldamento: foldr e le sue sorelle

La funzione **`foldl`** (= **fold left**): è lo schema ricorsivo necessario per spingere valori in avanti.

Possiamo scrivere la nostra funzione `reversEff` usando `foldl`.

Se la funzione foldata **è associativa e commutativa non fa differenza…**

sarebbe carino dimostrarlo con ragionamenti algebrici 😉

```
 -- notare che foldl accumula i conti sul
 -- suo secondo parametro:
myFoldL f v   []   = v
myFoldL f v (x:xs) = myFoldL f (f v x) xs

 -- da cui:
ReversEff' = myFoldL (\xs x->x:xs) []

 -- non fa differenza se la funzione foldata è
 -- associativa e commutativa…
mySum' = foldl (+) 0
mySum''= foldr (+) 0
```

Lazyness

Una interessante possibilità offerta da Haskell è quella della **valutazione lazy** (pigra) che ben si sposa con la strategia **call-by-name** del passaggio dei parametri.

Una delle applicazioni della valutazione lazy è la possibilità di maneggiare in modo molto astratto ed efficace **strutture dati infinite**, non altrettanto naturale coi linguaggi **eager**.

```
 -- Possiamo definire la lista con infiniti 1
ones = 1:ones
 -- Ovviamente, se chiediamo di valutare ones
 -- ci vengono stampati infiniti 1...
> ones
[1,1,1,1,1,1,1,1,1,1,…
^C
 -- tuttavia…
> take 3 ones
[1,1,1]
 -- … termina correttamente. take è predefinita
myTake  _    []    = []
myTake  0    xs    = []
myTake  n (x:xs) = x : myTake (n-1) xs
```

Valutazione Lazy di Espressioni

Cerchiamo di vedere come opera la valutazione lazy e riduciamo `take 3 ones`.

```
take 3 ones
      -- per ridurre take devo trasformare ones
      -- nella forma x:xs
    → take 3 (1:ones)
      -- riduco sempre il redex più esterno,
      -- applicando la clausola per take
    → 1:take 2 ones
      -- come prima: sono costretto a ridurre ones
    → 1:take 2 (1:ones)
      -- e così via...
    → 1:1:take 1 ones
    → 1:1:take 1 (1:ones)
    → 1:1:1:take 0 ones → 1:1:1:[]
 --  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
      -- Teniamo a mente la def di take:
myTake  _   []    = []
myTake  0   xs    = []
myTake  n (x:xs) = x : myTake (n-1) xs
```

Alcuni simpatici esempi…

Come definire lo **stream** (=lista infinita) dei numeri naturali?

Possiamo usare un generatore…

Oppure aiutarci con i nostri funzionali sulle liste…

Analogamente possiamo fare le potenze di un numero, ad esempio…

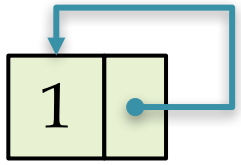
```
 -- Uso di un generatore (parametrico) per
 -- definire i naturali:
nats = nextNat 0 where
   nextNat n = n:nextNat (n+1)

 -- Forse più affascinante il seguente:
nats' = 0:map (+1) nats'

 -- Capito il trucco…
powers n = 1:map (n*) powers n
```

Definizioni Circolari

In una definizione come `ones` o `nats'`, il compilatore Haskell capisce che deve generare una lista infinita e durante la generazione è sufficiente **spostare un puntatore**.



Cosa che non gli è possibile per `powers` o `natGren` che dipendono da un parametro… tuttavia…

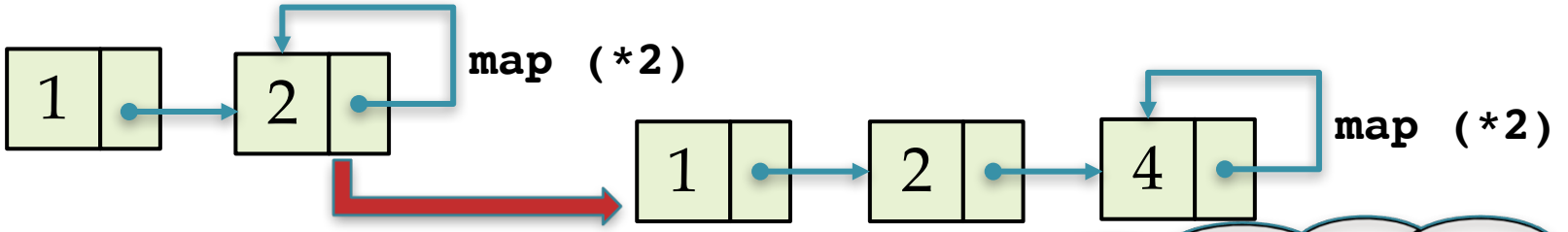
```
 -- Quindi è molto meglio:
powers' n = pws where pws = 1 : map (n*) pws

 -- Generalizziamo: non circolare
iter f x = x : map f (iter f x)
 -- oppure: circolare:
 -- il trucco è liberarsi dei parametri necessari
 -- in una definizione locale in cui sono `globali`
iterCirc f x = fs where fs = x : map f fs
```

Valutazione Lazy di Espressioni

Vediamo la differenza nella valutazione di `iter` e `iterCirc`.

Osservate che fs viene `prodotta` un elemento alla volta e il nostro processo **è sempre allineato all'ultimo elemento prodotto** quindi è sufficiente moltiplicare per 2 l'ultimo elemento generato!



produrre n elementi
è quadratico in n

```
iter (*2) 1
→ 1:map (*2) (iter (*2) 1)
→ 1:2:map (*2) (map (*2) (iter (*2) 1))
→ 1:2:4:map (*2) (map (*2)(map (*2) (iter (*2) 1)))

iterCirc (*2) 1
→ 1 : map (*2) fs
→ 1 : 2: map (*2) fs
→ 1 : 2: 4 : map (*2) fs
```

produrre n elementi
è lineare in n

Alcuni problemi con liste infinite

Attenzione alle definizioni circolari! Il processo di generazione **deve consumare meno input di quanto output produce**!

Questa equazione ricorsiva **è soddisfatta da ogni stream** e infatti non definisce niente! È inconsistente!

Attenti anche a list-comprehension su liste infinite!

```
 -- per esempio:
incstnts = (head incstnts):(tail incstnts)

factorsNT n = [x | x<-(tail nat'), n `mod` x == 0 ]
> factorsNT 24
[1,2,3,4,6,8,12,24^CInterrupted.
-- non termina perché cerca in tutti i naturali...
-- utile usare takeWhile che si ferma quando la
-- condizione è falsa
factors n = [x | x<-takeWhile (<n)(tail nat'),
        n `mod` x == 0 ]
> factors 24
[1,2,3,4,6,8,12,24]
primes = [p | p<-[2..], factors p = [1,p]]
```

Una piccola chicca per finire…

Definiamo lo stream dei numeri di **Fibonacci**: trucco considero due liste di fibonacci uno un passo avanti all'altra e sommo i numeri corrispondenti.

Visto che la definizione è circolare, in realtà esiste un'unica copia della lista `fibs`.

```
 -- Uso di un generatore (parametrico) per
 -- definire i naturali:
fibs = 0:1:zipWith (+) fibs (tail fibs)

 -- È più carino usare definizioni guardate,
 -- in cui non posso chiedere il tail e produco
 -- sempre almeno un elemento (ho mutua ricorsione)
fibs' = 0:fibs''
fibs''= 1:zipWith (+) fibs' fibs''

 -- da cui, sostituendo in fibs''
 --(anche da fibs, spingendo dentro l'uno...)
fibs'''= 0:zipWith (+) fibs''' (1:fibs''')
```

Lezione 6

That's all Folks…

Grazie per l'attenzione…

…Domande?