Ricorsione e iterazione

Mario Verdicchio

Funzioni computabili

- Una funzione f si dice computabile quando esiste un algoritmo che permette di calcolare, dato il valore x in input, il valore di y tale che y = f(x)
- Tutte le funzioni f: $N \rightarrow N$ computabili si ottengono a partire da un numero ristretto di funzioni base per mezzo di specifiche operazioni

Funzioni base

- Funzione zero

  $z: N \rightarrow N$, $z(x) = 0$ per ogni x
- Funzione successore

  $s: N \rightarrow N$, $s(x)$ è il successore di x
- Funzioni proiezione

  $P^n_i : N^n \rightarrow N$, $P^n_i(x_1,\ldots,x_n) = x_i$

Operazioni

- Composizione

  $h(x) = f \circ g(x) = f(g(x))$
- Ricorsione

  $n! = n \times (n-1)!$
- Minimalizzazione

  f(x) = il più piccolo y: g(x,y)=0

Costruzione di funzioni

- Con le operazioni di composizione e di ricorsione si costruiscono le cosiddette funzioni ricorsive primitive, che sono computabili e totali
- Se ricorriamo anche all’uso della minimalizzazione, si costruiscono le funzioni ricorsive generali, che sono sempre computabili, ma possibilimente parziali

Ricorsione in dettaglio

- Da f: $N^n \rightarrow N$ e g: $N^{n+2} \rightarrow N$ otteniamo la funzione h: $N^{n+1} \rightarrow N$ per ricorsione se:

$$\begin{cases} h(x_1,\ldots,x_n,0) = f(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n,s(y)) = g(x_1,\ldots,x_n,y,h(x_1,\ldots,x_n,y)) \end{cases}$$

caso base

passo

ipotesi ricorsiva

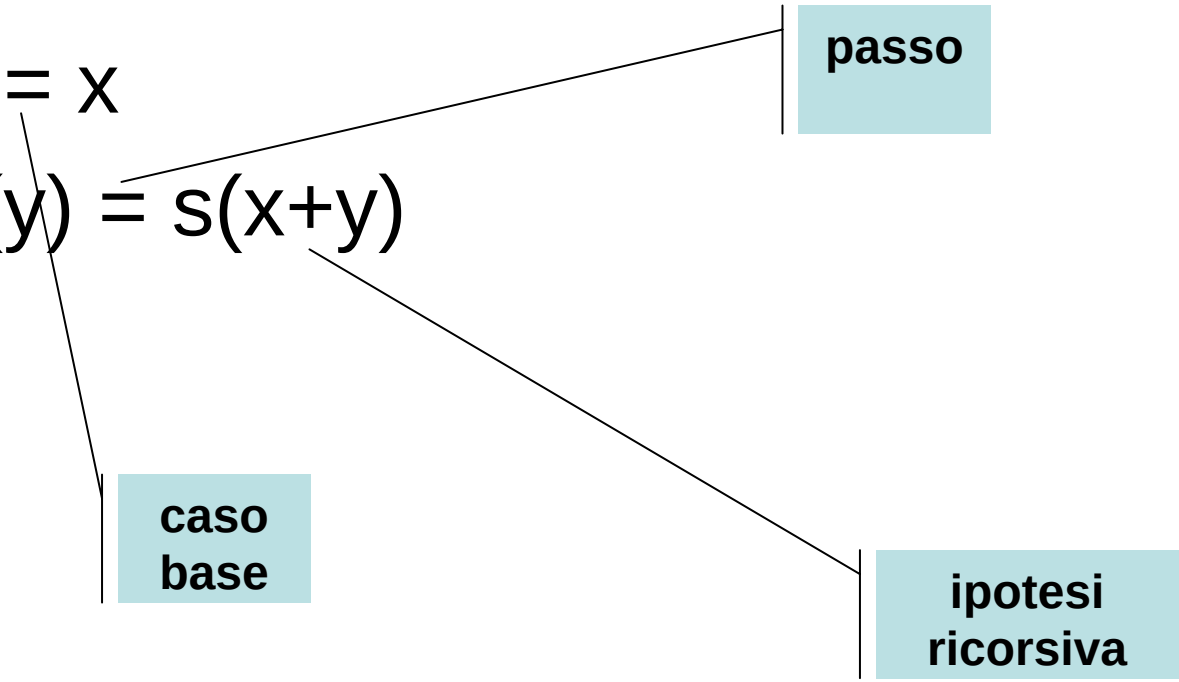
Esempio di costruzione di funzione

1. $f_1(x) = P^1_1(x)$ [funzione base]
2. $f_2(x) = s(x)$ [funzione base]
3. $f_3(x,y,z) = P^3_3(x,y,z)$ [funzione base]
4. $f_4(x,y,z) = s(P^3_3(x,y,z)) = s(z)$
   [composizione di f_2 e f_3]
5. $\begin{cases} f_5(x,0) = f_1(x) = x \\ f_5(x,s(y)) = f_4(x,y,f_5(x,y)) = s(f_5(x,y)) \end{cases}$
   [ricorsione su f_1 e f_4]

Esempio di costruzione: la somma

$$\begin{cases} f_5(x,0) = f_1(x) = x \\ f_5(x,s(y)) = f_4(x,y,f_5(x,y)) = s(f_5(x,y)) \end{cases}$$

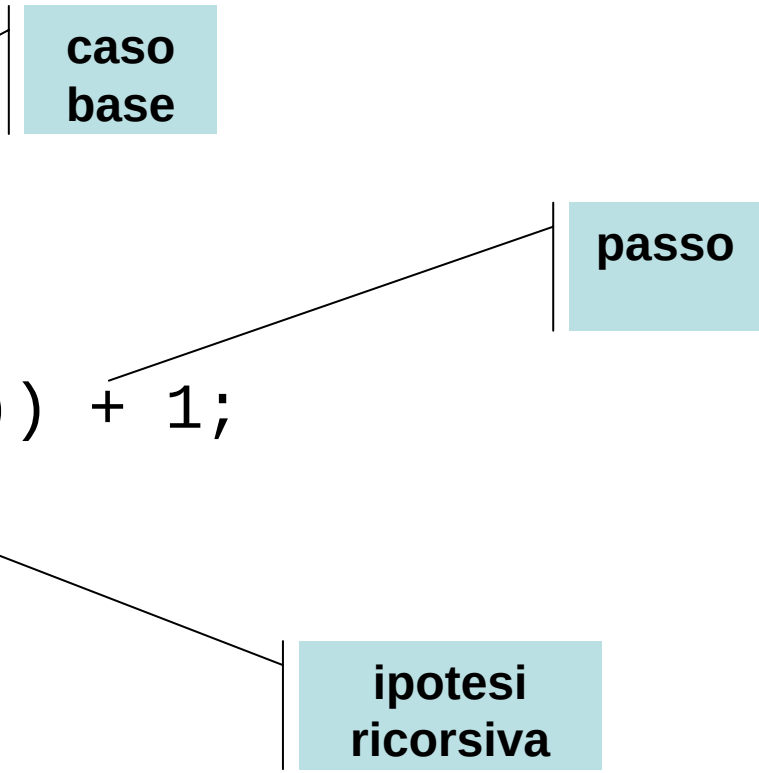
$$\begin{cases} x+0 = x \\ x+s(y) = s(x+y) \end{cases}$$



Funzioni definite con ricorsione

- `length` calcola la lunghezza di una stringa

```
int length (string s){
  if empty(s)
     return 0;
  return length(rest(s)) + 1;
}
```



Esecuzione

- Invocando `length` su “`abcd`” si ha:

1. `length(“bcd”) + 1`
2. `(length(“cd”) + 1) + 1`
3. `((length(“d”) + 1) + 1) + 1`
4. `(((length(“”) + 1) + 1) + 1) + 1`
5. `0 + 1 + 1 + 1 + 1 = 4`

- La lettera ‘`a`’ è l’ultima ad essere contata

Versione alternativa

```
int length (string s){
  return length_1(s,0);
}

int length_1 (string s, int x){
  if empty(s)
     return x;
  return length_1(rest(s),x+1);
}
```

Esecuzione

- Invocando `length` su “`abcd`” si ha:

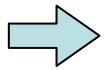
1. `length_1(“abcd”,0)`
2. `length_1(“bcd”,1)`
3. `length_1(“cd”,2)`
4. `length_1(“d”,3)`
5. `length_1(“”,4)`
6. `return 4`

- La lettera ‘`a`’ è la prima ad essere contata

La seconda versione è sostanzialmente una iterazione

```
int length (string s){
   return length_1(s,0);
}

int length_1(string s,
   int x){
   if empty(s)
       return x;
   return
   length_1(rest(s),x+1);
}
```



```
int length (string s){
   string s1 = s;
   int x;
   for(x=0;!empty(s1);x++)
       s1 = rest(s1);
   return x;
}
```

Diversi tipi di ricorsione

- Una ricorsione del primo tipo si dice propria
- Una ricorsione del secondo tipo si dice impropria, o iterazione in forma ricorsiva
- La differenza sta nel fatto che in una ricorsione propria la chiamata ricorsiva viene effettuata all'interno di un'altra funzione (nel caso di `length`: la somma)

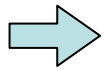
La ricorsione propria in generale

```
int funzione(int s){
  if condizione(s)
     return e;
  return h(funzione(g(s)),k(s));
}
```

La ricorsione impropria in generale

```
int funz(int s){
   return funz_1(s,e);
}

int funz_1(int s,int x){
   if cond(s)
       return x;
   return
   funz_1(g(s),h(k(s),x));
}
```

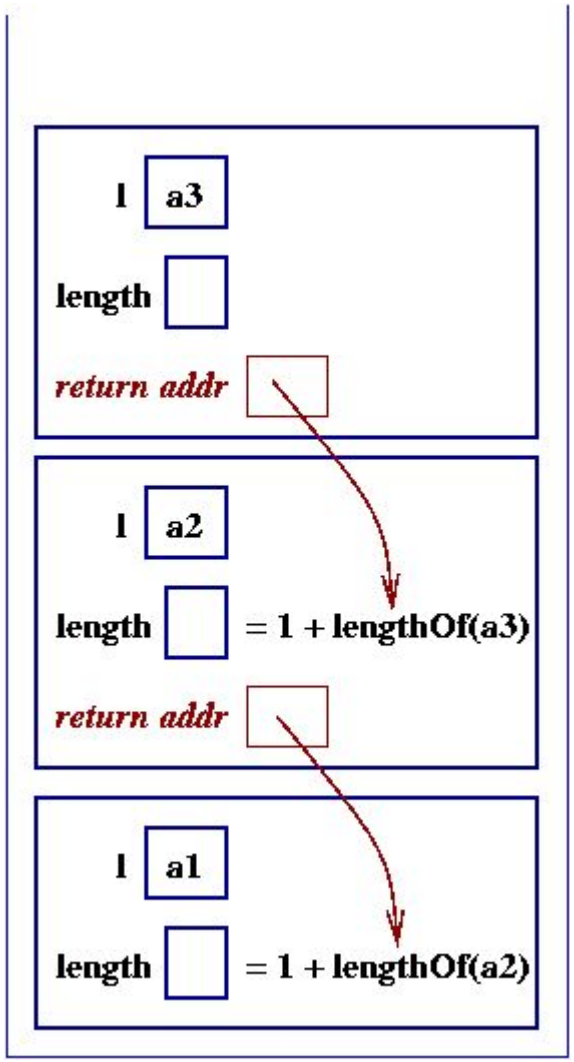


```
int funz(int s){
   int s1 = s;
   int x;
   for(x=e;!cond(s1);x=h(k(s1),x))
       s1 = g(s1);
   return x;
}
```

Ricorsione vs. iterazione

- La ricorsione in molti casi fornisce una soluzione elegante e rapida da codificare
- L'iterazione è però senza dubbio più efficiente dal punto di vista delle risorse di calcolo in fase di esecuzione
- La `length` ricorsiva su una stringa lunga n comporta n chiamate a funzione, mentre nella versione iterativa si ha una sola chiamata

Activation stack



Quindi...

- Se si ha una funzione implementata con una ricorsione impropria, è una buona scelta trasformarla in una iterazione
- E se si ha una funzione implementata con una ricorsione propria?
- Nel caso di `length` è stata trovata la versione impropria. Ma è sempre possibile?

La funzione length: da ricorsione propria a impropria

```
int length (string s){
   if empty(s)
       return 0;
   return length(rest(s))
          + 1;
}
```

```
int length (string s){
   return length_1(s,0);
}

int length_1(string s,
   int x){
   if empty(s)
       return x;
   return
   length_1(rest(s),x+1);
}
```

In generale

```
int funz (int s){
  if cond(s)
      return e;
  return
    h( funz(g(s)),k(s) );
}
```

```
int funz(int s){
  return funz_1(s,e);
}

int funz_1(int s,int x){
  if cond(s)
      return x;
  return
  funz_1(g(s),h*(k(s),x));
}
```

Questa trasformazione è possibile se

$h(e_1,h(e_2,...h(e_n,e)...)) =$

$h^*(e_n,h^*(e_{n-1},...h^*(e_1,e))...))$

Altro esempio: le liste

- Sia dato il tipo di dato list (liste di interi), su cui sono definite le seguenti funzioni:

```
bool empty(list l);
int first(list l); //first((1 2 3)) == 1
list rest(list l); //rest((1 2 3)) == (2 3)
list cons(int n, list l);
//cons(0, (1 2 3)) == (0 1 2 3)
```

Funzione reverse

```
list reverse(list l);
// reverse((1 2 3 4)) == (4 3 2 1)
```

- Implementare la funzione in versione ricorsiva propria, e poi, se possibile, in versione ricorsiva impropria, e quindi iterativa