Progettazione ricorsiva co-variante

Per programmare sia ricorsivamente, sia iterativamente, occorre avere ben chiaro il risultato che vogliamo ottenere dall'algoritmo.

Come guida sviluppiamo due casi concreti di progettazione di algoritmi ricorsivi co-varianti per *evidenziare lo schema che li accomuna* ed i cui principi sono applicabili tutte le volte che occorre progettare un algoritmo, sia esso ricorsivo co-variante, sia esso iterativo.

Sottrazione

Supponiamo di voler calcolare la differenza tra due numeri naturali `x` e `y`, tali che `y <= x`. Lo scopo è scrivere un algoritmo `meno(x,y)` che restituisce un numero che corrisponde alla differenza tra i valori `x` e `y`. Quindi, la proprietà, o predicato, che vogliamo soddisfare è `meno(x,y) == x - y`.

Leggiamo il significato di tale predicato:

1. l'algoritmo `meno(x,y)` deve produrre un valore che chiamiamo a e che dipenderà da `x` e `y`;
2. `x-y` è un altro valore, che chiamiamo b, anch'esso dipendente da `x` e `y`;
3. richiediamo che `meno(x,y)` funzioni in modo a e b siano uguali, cioè che a==b; siccome a è il valore di `meno(x,y)` e b quello di `x - y`, stiamo appunto richiedendo che il predicato `meno(x,y) == x - y` sia vero.

Per progettare l'algoritmo `meno(x,y)` è utile porsi alcune domande.

1ma domanda (e risposta) *Con quali valori di* `x` *e* `y` *è ovvio (immediato, non costa alcuno sforzo) calcolare il valore* `x-y`?

Per rispondere, basta osservare che, se `y` è il valore `0`, il valore della differenza tra un qualsiasi `x` e `0` è `x`. Quindi `meno(x,0)` non deve far altro se non restituire `x`; la definizione di `meno(x, 0)` diventa:

```
meno(x, 0) = x
```

che si legge: "quando il secondo argomento dell'algoritmo di nome `meno` vale `0`, il risultato è il primo argomento".

2da domanda (e risposta) Poniamoci ora l'obiettivo di stabilire come si deve comportare l'algoritmo `meno(x,y)`quando, per un dato valore `x`, il valore di `y` è maggiore di `0`.

La domanda fondamentale per arrivare alla sua definizione è:

Quale algoritmo risolve un problema appena più semplice di quello risolto da `meno(x,y)` *di cui sto cercando la definizione?*

Per rispondere, ragioniamo *induttivamente*:

1. Ipotizziamo di saper definire il comportamento dell'algoritmo `meno(x,y-1)`, cioè dell'algoritmo che risolve la sottrazione tra `x` ed un valore appena più piccolo di `y`, cioè di `y-1`;
2. Se `meno(x,y-1)` si comporta secondo le attese, *per ipotesi induttiva* esso soddisfa la relazione `meno(x,y-1) == x-(y-1) == x-y+1`;
3. Se riscrivo l'equazione `meno(x,y-1) == x-y+1`, togliendo una unità sia all'espressione sulla sinistra di `==`, sia all'espressione sulla destra, ottengo:

   ```
   meno(x,y-1)-1 == x-y+1-1 == x-y
   ```

Leggiamo l'espressione `meno(x,y-1)-1 == x-y` appena scritta:

"*Il valore* `x-y` *è ottenuto togliendo una unità dal valore fornito dall'algoritmo* `meno(x,y-1)`".

Ma `x-y` è il valore che vogliamo ottenere proprio dall'algoritmo `meno(x,y)` di cui stiamo cercando la definizione. Quindi, possiamo definire `meno(x,y)` sfruttando l'osservazione appena fatta:

```
meno(x,y) == meno(x,y-1) - 1
```

Conclusione Per ogni `x` ed `y` numeri naturali in cui `y <= x`, l'algoritmo cercato è definito per casi come segue:

```
meno(x,0) = x                   // caso base
meno(x,y) = meno(x,y-1) - 1     // caso induttivo
```

Vedremo che per il ***Principio di Induzione***, la definizione assicura che `meno(x,y)` produce un risultato corretto, cioè `meno(x,y) == x - y`, **per ogni numero** naturale `x` e `y` , a patto che `y <= x`.

Definizioni alternative ed *equivalenti* a quella per casi appena data, ma più vicine al codice del linguaggio di programmazione di riferimento, sono:

```
// versione che comincia col caso base
meno(x, y) {
  if (y == 0) { // caso base
                risultato = x
                return risultato
  } else { // caso induttivo
         risultatoInduttivo = meno(x, y - 1)
         risultato = risultatoInduttivo - 1
         return risultato
  }
}
// versione che comincia col caso induttivo
meno(x, y) {
  if (y > 0) { // caso induttivo
               risultatoInduttivo = meno(x, y - 1)
```

```
            risultato = risultatoInduttivo - 1
            return risultato
  } else { // caso base
         risultato = x
         return risultato
  }
}
```

Esperimento

Una volta sintetizzato l'algoritmo `meno(x,y)`, verifichiamo "sperimentalmente" il funzionamento del ragionamento induttivo, ad esempio, applicandolo passo passo al calcolo del valore `meno(x, 2)`, per un qualsiasi valore di `x >= y`:

1. Dovendo calcolare `meno(x,2)`, mi chiedo quale sia un problema appena più semplice da risolvere. Secondo quanto descritto in precedenza, il problema `meno(x,1)` è più semplice: il valore `1` è più prossimo a `0` di quanto non lo sia `2`.

2. Quanto vale `meno(x,1)`? Per rispondere, mi chiedo quale sia un problema più semplice da risolvere. Sicuramente `meno(x,0)` è più semplice di `meno(x,1)` perché conosco il risultato di `meno(x,0)`, che è banalmente `x`.

3. Ora che ho ottenuto `meno(x,0) == x` posso ricavare:

   ```
   meno(x,1) == meno(x,0) - 1 == x - 1
   ```

4. Ora che ho ottenuto `meno(x,1) == x - 1` posso ricavare:

   ```
   meno(x,2) == meno(x,1) - 1 == (x - 1) - 1 == x - 2
   ```

   che è il risultato cercato.

Perché la parola "co-variante"?

La definizione:

```
meno(x,0) = x                  // caso base
meno(x,y) = meno(x,y-1) - 1   // caso induttivo
```

è **co-variante** perché il valore dell'argomento `y` che *individua il caso da applicare*, cioè la *variabile induttiva che guida lo svilupparsi della ricorsione*, diminuisce col diminuire della "difficoltà" (dimensione) del problema da risolvere.

Nel nostro caso, `meno(x,y-1)` è più semplice da risolvere di quanto è `meno(x,y)` perché `meno(x,y-1)` è più vicino al caso base `meno(x,0)`.

Moltiplicazione

Supponiamo di voler calcolare il prodotto tra due numeri naturali `x` e `y`. Lo scopo è scrivere un algoritmo `molt(x,y)` che restituisce un numero che corrisponde al prodotto tra i valori `x` e `y`. Quindi, la proprietà che vogliamo soddisfare è `molt(x,y) == x * y`.

Leggiamo il significato di tale predicato:

1. l'algoritmo `molt(x,y)` deve produrre un valore, che chiamiamo a e che dipenderà, ovviamente da `x`e `y`;
2. `x*y` è un altro valore, che chiamiamo b, anch'esso dipendente da `x`e `y`;
3. richiediamo che `mol(x,y)` funzioni in modo che i valori a e b siano uguali, cioè che a==b; siccome a è il valore di `molt(x,y)` e b quello di `x * y`, stiamo appunto richiedendo che il predicato `molt(x,y) == x * y` sia vero.

Per progettare l'algoritmo `molt(x,y)` è utile porsi alcune domande.

1ma domanda (e risposta) *Con quali valori di* `x` *e* `y` *, è ovvio (immediato, non costa alcuno sforzo) calcolare il valore* `x*y`?

Per rispondere, basta osservare che, se `y` è il valore `0`, il valore del prodotto tra un qualsiasi `x` e `0` è ovviamente `0`. Quindi `molt(x,0)` non deve far altro se non restituire `0`; la definizione di `molt(x, 0)` diventa:

```
molt(x, 0) = 0
```

che si legge come: "quando il secondo argomento dell'algoritmo di nome `molt` vale `0`, il risultato è necessariamente `0`".

2da domanda (e risposta) Poniamoci ora l'obiettivo di stabilire come si deve comportare l'algoritmo `molt(x,y)`quando, per un dato valore `x`, il valore di `y` è maggiore di `0`.

La domanda fondamentale per arrivare alla sua definizione è:

Quale algoritmo risolve un problema appena più semplice di quello risolto dall'algoritmo `molt(x,y)` *di cui sto cercando la definizione?*

Per rispondere, ragioniamo *induttivamente*:

1. Ipotizziamo di saper definire il comportamento dell'algoritmo `molt(x,y-1)`, cioè dell'algoritmo che risolve la moltiplicazione tra `x` ed un valore appena più piccolo di `y`, cioè di `y-1`;

2. Se `molt(x,y-1)` si comporta secondo le attese, *per ipotesi induttiva* esso soddisfa la relazione `molt(x,y-1) == x*(y-1) == x*y-x`;

3. Se riscrivo l'equazione `molt(x,y-1) == x*y-x`, sommando `x` sia all'espressione sulla sinistra di `==`, sia all'espressione sulla destra, ottengo:

   ```
   molt(x,y-1)+x == x*y-x+x == x*y
   ```

Leggiamo l'espressione `molt(x,y-1)+x == x*y` appena scritta:

"*il valore* `x*y` *è ottenuto sommando* `x` *al valore fornito dall'algoritmo* `molt(x,y-1)`".

Ma `x*y` è il valore che vogliamo ottenere proprio dall'algoritmo `molt(x,y)` di cui stiamo cercando la definizione. Quindi, possiamo definire `molt(x,y)` sfruttando l'osservazione appena fatta:

```
molt(x,y) == molt(x,y-1) + x
```

Conclusione Per ogni `x` ed `y` numeri naturali, l'algoritmo cercato è definito per casi come segue:

```
molt(x,0) = 0                // caso base
molt(x,y) = molt(x,y-1) + x  // caso induttivo
```

Vedremo che per il ***Principio di Induzione***, la definizione assicura che `molt(x,y)` produce un risultato corretto, cioè `molt(x,y) == x * y`, **per ogni numero** naturale `x` e `y`.

Definizioni alternative ed *equivalenti* a quella per casi appena data, ma più vicine al codice del linguaggio di programmazione di riferimento sono:

```
// versione che comincia col caso base
molt(x, y) {
  if (y == 0) { // caso base
                risultato = 0
                return risultato
  } else { // caso induttivo
         risultatoInduttivo = molt(x, y - 1)
         risultato = risultatoInduttivo + x
         return risultato
  }
}
// versione che comincia col caso induttivo
molt(x, y) {
  if (y > 0) { // caso induttivo
               risultatoInduttivo = molt(x, y - 1)
               risultato = risultatoInduttivo + x
               return risultato
  } else { // caso base
         risultato = 0
         return risultato
  }
}
```

Esperimento

Una volta sintetizzato l'algoritmo `molt(x,y)`, verifichiamo "sperimentalmente" il funzionamento del ragionamento induttivo, ad esempio, applicandolo passo passo al calcolo del valore `molt(x, 2)`, per un qualsiasi valore di `x`:

1. Dovendo calcolare `molt(x,2)`, mi chiedo quale sia un problema appena più semplice di esso da risolvere. Secondo quanto descritto in precedenza, il problema `molt(x,1)` è più semplice: il valore `1` è più prossimo a `0` di quanto non lo sia `2`;
2. Ma quanto vale `molt(x,1)`? Per rispondere, mi chiedo quale sia un problema più semplice da risolvere. Sicuramente `molt(x,0)` è più semplice di `molt(x,1)` perché conosco il risultato di `molt(x,0)`, che è banalmente `0`.
3. Ora che ho ottenuto `molt(x,0) == 0`, posso ricavare:

   ```
   molt(x,1) == molt(x,0) + x == 0 + x
   ```

4. Ora che ho ottenuto `molt(x,1) == 0 + x`, posso ricavare:

   ```
   molt(x,2) == molt(x,1) + x == (0 + x) + x == x + x
   ```

   che è il risultato cercato.

Perché la parola "co-variante"?

La definizione:

```
molt(x,0) = x                // caso base
molt(x,y) = molt(x,y-1) + x  // caso induttivo
```

è **co-variante** perché il valore dell'argomento `y` che *individua il caso da applicare*, cioè *la variabile induttiva che guida lo svilupparsi della ricorsione*, diminuisce col diminuire della "difficoltà" (dimensione) del problema da risolvere.

Nel nostro caso, `molt(x,y-1)` è più semplice da risolvere di quanto sia `molt(x,y)` perché `molt(x,y-1)` è più vicino al caso base `molt(x,0)`.