Programmazione Funzionale: Principi

Sommario: 23 Maggio, 2018

- Programmazione Funzionale: Principi e Proprietà
  - > Modello di Calcolo
  - > Trasparenza Referenziale
- Strutture Fondamentali:
  - > Espressione, Applicazione, Strategia di Valutazione
  - > Sistema di Tipi e Polimorfismo Generico
  - > API-ADT: Modulo Segnature, Moduli Implementazione
- Higher Order:
  - > Principio di Astrazione e Riuso di Codice
  - > Funzioni Higher Order: List.fold_left
  - > List.fold_right
  - > List.map
  - > List.filter
  - > Definizioni Tail Recursive

Programmazione Funzionale: Principi e Proprietà

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
  - Nessuna Memoria [1]: Valori Non Modificabili
  - Trasparenza Referenziale:[2]

Definition

Sia $\mathtt{P}$ un programma ed $\mathtt{E_i}$ un suo termine (espressione) all'occorrenza $\mathtt{i}$. Se la computazione di $\mathtt{P}$ riduce $\mathtt{E_i}$ al valore $\mathtt{V_{E_i}}$, allora il programma $\mathtt{P}[\mathtt{E_i} \leftarrow \mathtt{V_{E_i}}]$ è equivalente a $\mathtt{P}$ e può rimpiazzarlo in ogni computazione.

Ad esempio. `f(x)+f(x)` $\equiv$ 2 * f(x), con + somma, e * prodotto.

  - Correttezza e Verifica: Trasparenza Referenziale permette tecniche di prova piuttosto semplici.

[1] intesa come stato

[2] $\mathtt{P}[\mathtt{E} \leftarrow \mathtt{V_E}]$ indica il rimpiazzamento di ogni occorrenza di $\mathtt{E}$ in $\mathtt{P}$ con $\mathtt{V_E}$

Programmazione Funzionale: Principi e Proprietà /2

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
  - Nessuna Memoria
  - Trasparenza Referenziale
  - Correttezza e Verifica
  - Funzioni sono valori:
    - calcolabili da un'applicazione di funzione
    - passabili come argomenti a un'applicazione di funzione
  - Programmazione (astratta e) Higher Order
    - Astrazioni sui dati: Modellare valori con funzioni [3]
    - Astrazioni sul controllo: Modellare strutture di controllo con funzioni [4]
  - Computazione per Riduzione: Come per il Lambda-Calcolo (vedi Esercizio su SOS per Lambda-Calcolo)

[3] ad es. interi di Barendregt, env e store in Laboratorio 2018

[4] ad es. iteratori fold_left e fold_right in OCaml

Strutture Fondamentali: Funzioni, Applicazioni, Espressioni

- **Programmi** sono:
  - Una struttura (di Moduli) con definizioni di funzioni e tipi di dato, che forma un ambiente (di bindings)
  - Una espressione da valutare in tale ambiente
- **Funzioni**:
  - Sono definite da espressioni:
    Ad es.: (OCaml) `fun x` $\rightarrow$ `let...in exp`
  - Possono contenere blocchi-espressione, `let_in_`
  - Sono Programming Units: Il corpo ha la stessa struttura del programma:
- **Applicazione**: Il Meccanismo di calcolo fondamentale
- **Trasmissione e Strategia di Valutazione**:
  - OCaml: Trasmissione per valore
  - Haskell: Valutazione esterna e Lazy costruttori

Strutture Fondamentali: Tipi

- Sistema dei Tipi (Pure OCaml).
  - **Tipi Basici Scalari**: int, float, bool, char, e (char)string
  - **T. Polimorfi e Variabili di tipo**. Ad es.: `'a`
  - **Tipi Basici Strutturati**: Tuples, List polimorfe (generiche).
    Ad es.: `('a * int) list`
  - **Tipi funzione**: $\rightarrow$.
    Ad es.: $'a$ `list` $\rightarrow$ ($'a \rightarrow' a \rightarrow$ `bool`) $\rightarrow' a$ `list`
  - **Tipi Concreti**: Record e Variant Types.
    - **Definiti** mediante costrutto:
      type typeName = typeExpression
    - **Record**. Ad es.:
      `type ratio = {num:int; denum:int}`
    - **Variant o Algebrici**. Ad es.:
      `type ('a,'b) myType = Either of 'a | Or of 'b;;`
    - **Variant Ricorsivi**. Ad es.:
      `type 'a myList = Nil | Cons of 'a * 'a myList;;`
  - **Tipi Astratti**

Tipi Astratti: API o Segnatura

- Modulo Signature: ha un nome,
- Contiene la signature dei pubblici
- La signature è racchiusa tra **sig**...**end**
- Elenca i tipi esportati (e implementati dall'ADT)
- Elenca la segnatura di ogni operazione esportata
- La corrispondenza tra API e ADT è controllata dal sistema dei tipi.

```
module type RELAZIONE =
 sig
  type ('a,'b) relazione
  val relazioneC: unit -> ('a,'b) relazione
  val addPair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val removePair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val getUno: ('a,'b) relazione -> 'b -> 'a list
  val getDue: ('a,'b) relazione -> 'a -> 'b list
end;;
```

Tipi Astratti: ADT e Implementazione

- Modulo ADT: Ne possono esistere più d'uno.
- Ogni ADT ha un nome,
- Contiene le implementazioni
- L'implementazione è racchiusa tra **struct**...**end:NomeSign.**
- Implementazione tipi esportati
- Implementazione operazioni esportate

```
module Relazione =
 (struct
  type ('a,'b) relazione = ('a * 'b) list
  let relazioneC = fun () -> []
  let removePair r x y =
     let g = fun u -> if u=(x,y) then [u] else []
     in List.fold_right(List.append)(List.map g r)[]
  let isIn r v = List.fold_right (||) (List.map ((=)v) r) false
  let addPair r x y = if (isIn r (x,y)) then r else (x,y)::r
  let isDue(r,y) = List.fold_right (||) (List.map(fun u -> snd(u)=y) r) false
  let getUno r y = let g = fun u ->if(snd(u)=y)then[fst(u)]else[]
                   in List.fold_right(List.append)(List.map g r) []
  let getDue r x = let g = fun u ->if(fst(u)=x)then[snd(u)]else[]
                   in List.fold_right(List.append)(List.map g r) []
end:RELAZIONE);;
```

Tipi Astratti: Quali Additionals?

- **Anatomia di ADT OCaml**
  Nessuna differenza con quella di ADT Java

  - **Stato Concreto c:** Struttura Implementazione dei Va [5]
    - `type ('a,'b) relazione = ('a*'b) list`
    - In generale, una tupla (o record) di tipi, o algebrici
      type tree = {label: labelType; sons: tree list}
      type tree = Tree of labelType * tree list
  - **AF e I:** Funzione di Astrazione e Invariante
    - $\texttt{AF(c)} = \{(\texttt{x},\texttt{y}) \mid (\exists \texttt{n} \in [\texttt{0..length(c)} - \texttt{1}])\ \texttt{hd(tl}^{\texttt{n}}\ \texttt{c)} == (\texttt{x},\texttt{y})\}$ [6]
  - **Additionals:** Ma per valori senza stato
    - toString: Stringa di presentazione del Va
      $\texttt{toString} : (\texttt{'a,'b})\texttt{relazione} \rightarrow \texttt{String}$
    - elements: Lista degli elementi contenuti nel Va
      $\texttt{elements}(\texttt{'a,'b})\texttt{relazione} \rightarrow (\texttt{'a} * \texttt{'b})\texttt{list}$

[5] Va indica un valore astratto
[6] $\texttt{g}^{\texttt{n}}$ indica n composizioni di g

Riuso e Principio di Astrazione

- Una prima definizione:

```
let rec addAll acc = function
  |[ ] -> acc
  |x::xs -> addAll (acc+x) xs;;
```

- Una seconda definizione:

```
let rec rev acc = function
  |[ ] -> acc
  |x::xs -> rev (x::acc) xs;;
```

- Hanno identica forma (pattern)

Definition (Principio di Astrazione)

Evitare programmi che richiedono di dichiarare più volte una "stesso pattern di controllo". Le similarità devono essere individuate e astratte creando opportune funzioni che le implementino.

List.fold_left e Funzioni Higher Order

- Un'unica dichiarazione:

```
let rec fold_left g acc = function
  |[ ] -> acc
  |x::xs -> fold_left g (g acc x) xs;;
```

- Tanti usi diversi:

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
#addAll0 [1;2;3;4;5];;
- :  int = 15
#revT [1;2;3;4;5];;
- :  int list = [5; 4; 3; 2; 1]
```

Funzioni Higher Order

- Una funzione Higher Order ha funzioni come parametro: `g`

```
let rec fold_left g acc = function
  |[ ] -> acc
  |x::xs -> fold_left g (g acc x) xs;;
```

- oppure, calcola una funzione: `addAll,rev,addAll0,revT`

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
```

- oppure, entrambe le cose:

```
fold_left g
```

List.map

- Ancora pattern simili:

```
let rec squareEach = function
  |[ ] -> [ ]
  |x::xs -> (x * x)::(squareEach xs);;
let rec show = function
  |[ ] -> [ ]
  |x::xs -> (x^", ")::(show xs);;
```

- Un'unica funzione:

```
let rec map g = function
  |[ ] -> [ ]
  |x::xs -> (g x)::(map g xs);;
```

- tanti usi diversi:

```
let squareEach = map (fun x -> x*x);;
let show = map (fun x -> x^", ");;
```

List.filter

- Ancora pattern simili:

```
let rec pos = function
  |[ ] -> [ ]
  |x::xs -> if x >= 0 then x::(pos xs) else pos xs;;
let rec even = function
  |[ ] -> [ ]
  |x::xs -> if x mod 2 == 0 then x::(even xs) else even xs;;
```

- Un'unica funzione:

```
let rec filter g = function
  |[ ] -> [ ]
  |x::xs -> if g x then x::(filter g xs) else filter g xs;;
```

- tanti usi diversi:

```
let pos = filter (fun x -> x >= 0);;
let even = filter (fun x -> x mod 2 == 0);;
```

List.fold_right

- Ancora pattern simili:

```
let rec sub fin = function
  |[ ] -> fin
  |x::xs -> x - (sub xs);;
let rec toString fin = function
  |[ ] -> fin
  |x::xs -> x ^ (toString xs);;
```

- Un'unica funzione: (un'astrazione un più strana in OCaml)

```
let fold_right g xx fin = match xx with
  |[ ] -> fin
  |x::xs -> g x (forld_right g xs fin);;
```

- tanti usi diversi:

```
let sub fin = fun xx -> fold_rigth (-) xx fin;;
let toString fin = fun xx -> fold_right (^) xx fin;;
```

Riuso, Efficienza, Definizioni Tail Recursive

- **Astrazione di Pattern**. Invece di tante dichiarazioni `g1,...,gn` che instanziano in modo differente uno stesso pattern, nel programma si fa uso di una sola funzione `G` higher order, avente pattern che astrae ciascuna di tali istanze e si rimpiazza ogni `gi` con un'instanza diversa di `G` e specifica per `gi`.

- **Proprietà**. Proprietà provate sul pattern `G` possono essere opportunamente riformulate ed utilizzate in ogni sua instanza `gi`.
  - **Proviamo**. $\forall$ `(g:'a->'b->'b, xx:'a list, fin:'b)`
    `fold_right g xx fin == fold_left (Ex g) fin (rev xx)`[7]
    dove: `Ex h = fun a x -> h x a`

- **Efficienza**. Una ri-definizione più efficiente di `G` conduce a una corrispondente maggiore efficienza di ciascun instanza fornita per `g1,...,gn`, fornendo possibilmente pattern di calcolo alternativo e più efficiente per ognuna di tali funzioni `g1,...,gn`.
  - **Tail Recursive**. Mostriamo che le funzioni higher order, `fold_rigth`, `map`, `filter` hanno (un pattern tail recursive e quindi) una definizione tail recursive (in slide "folder_right è tail recursive" e successive).

[7] Una dimostrazione é nella slide successiva e rev indica la funzione che calcola la reverse di lista (definita anche in slide "Riuso e Principio di Astrazione")

Possiamo esprimere fold_right con fold_left

- **Lemma1** ∀ ((g:'a->'b->'b),(z:'a),(fin:b))
  g z fin == fold_left(E g) fin [z]
- **Lemma2** ∀ (g:'a->'b->'b, xx,yy:'a list, fin:'b)
  fold_left g fin (xx@yy) == fold_left g (fold_left g fin xx) yy
- **Lemma3** ∀ ((x:'a),(xx:'a list)) [x]@xx == x::xx
- **Lemma4** ∀ (x:'a), rev[x] == [x] [8]
- **Lemma5** ∀ (xx:'a list, yy:'a list),
  rev(xx@yy) == (rev yy)@(rev xx)

- **Proviamo**. ∀ (g:'a->'b->'b, xx:'a list, fin:'b)
  fold_right g xx fin == fold_left (Ex g) fin (rev xx)
  dove: Ex h = fun a x -> h x a
- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:
  ∀ (g:'a->'b->'b, zz:'a list, fin:'b) && List.length zz < k,
  fold_right g xx fin == fold_left (Ex g) fin (rev xx)
  - Sia xx == z::zz.

[8] rev indica la funzione che calcola la reverse di lista (definita anche in slide "Riuso e Principio di Astrazione")

La dimostrazione

- **L1.** `g z fin == fold_left(E g) fin [z]`
- **L2.** `fold_left g fin (xx@yy) == fold_left g(fold_left g fin xx)yy`
- **L3.** `[x]@xx == x::xx`
- **L4.** `rev[x] == [x]`
- **L5.** `rev(xx@yy) == (rev yy)@(rev xx)`
- **Proviamo**. ∀ `(g:'a->'b->'b, xx:'a list, fin:'b)`
  `fold_right g xx fin == fold_left (Ex g) fin (rev xx)`
  dove: `Ex h = fun a x -> h x a`
- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:
  ∀ `(g:'a->'b->'b, zz:'a list, fin:'b) && List.length zz < k`,
  `fold_right g xx fin == fold_left(Ex g) fin (rev xx)`
  - Sia `xx == z::zz`.
  - `fold_right g (z::zz) fin`
  - `== g z (fold_right g zz fin)` — `fold_right`
  - `== g z (fold_left(E g)fin(rev zz))` — ind. ass.
  - `== fold_left(E g)(fold_left(E g)fin(rev zz))[z]` — Lemma1
  - `== fold_left(E g)fin((rev zz)@[z])` — Lemma2
  - `== fold_left(E g)fin((rev zz)@(rev[z]))` — Lemma4
  - `== fold_left(E g)fin(rev([z]@zz))` — Lemma5
  - `== fold_left(E g)fin(rev(z::zz))` — Lemma3

Efficienza: Definizioni Tail Recursive

- **fold_right**

```
fold_right = fun g xx fin -> fold_left(Ex g) fin (rev xx)
```

  Usate: `fold_left`, `rev` hanno definizioni Tail Recursive.

- **map**

```
map = fun g xx -> fold_right (fun x a -> (g x)::a) xx [ ]
```

  Usate: `fold_rigth` ha definizione Tail Recursive.

- **filter**

```
filter = fun p xx ->
        let g = fun x a -> if (p x) then x::a else a in
        fold_right g xx [ ]
```

  Usate: `fold_right` ha definizione Tail Recursive.

Esercizi

- Perchè nella programmazione Funzionale `List.fold_left` è considerato un iteratore? Si spieghi cosa è valutato ad ogni iterazione e come variano gli eventuali indici dell'iterazione.
- (a) Cosa calcola `map (+5) (map (*2) xx)`?
  (b) Cosa si può dire in generale, su un pattern della forma: `map f (map g xx)`?
- (a) Si dia una definizione di `length` che calcola la lunghezza di una lista
  (b) Sapreste dare una definizione Tail Recursive di `length`
- Sia indList una funzione he applicata ad una arbitraria lista calcola la lista di coppie avente come primo componente la posizione della coppia nella lista e come secondo componente il valore in quella posizione della lista data. Ad esempio indList [56;12;-7] deve calcolare [(1,56),(2,12),(3,-7)].
  (a) Si dia una definizione di indList;
  (b) Si dia una definizione iterativa e Tail Recursive di indList che utilizzi le sole operazioni: `List.fold_left, fst, snd, List.length`.