

Sommario: 17 Maggio, 2017

- Programmazione Funzionale: Principi e Proprietà
- Strutture Fondamentali: Funzioni, Applicazioni, Espressioni, Tipi (Polimorfi)
- API-ADT: Modulo Segnature, Moduli Implementazione

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

Definition

Sia P un programma ed E un suo termine (espressione). Se la computazione di P valuta E al valore primitivo (o chiuso) V_E , allora il programma $P[E \leftarrow V_E]$ è equivalente a P e può rimpiazzarlo in ogni computazione.

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

- Correttezza e Verifica: Trasparenza Referenziale permette tecniche di prova

¹intesa come stato

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

- **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)

³ad es. interi di Barendregt, env e store in Laboratorio 2017

⁴ad es. iteratori `fold_left` e `fold_right` in OCaml 

- **Programmi** sono:
 - una struttura (di Moduli) con definizioni di funzioni e tipi di dato, che forma un ambiente (di bindings)
 - un'espressione da valutare in tale ambiente
- **Funzioni:**
 - sono definite da espressioni:
Ad es.: (OCaml) `fun x → 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.** Ad es.:
type ('a,'b) myType = Either of 'a | Or of 'b;;
 - **V. 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 isnl r v = List.fold_right (||) (List.map ((=)v) r) false
    let addPair r x y = if (isnl 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);;
```

- **Anatomia di ADT OCaml**

Nessuna differenza con quella di ADT Java

- **Stato Concreto c:** Struttura Implementazione dei Va⁵
 - `type ('a,'b) relazione = ('a,'b) list`
 - In generale, una tupla (o un record) di tipi
(vedi `Relazione2` in file `Relazione.ml` allegato 16/5/2017)
- **AF e I:** Funzione di Astrazione e Invariante
 - $AF(c) = \{(x, y) \mid (n \in [0..length - 1])hd(tl^n c) = (x, y)\}$ ⁶
- **Additional:** Ma per valori senza stato
 - `toString`: Stringa di presentazione del Va
`toString : ('a,'b)relazione → String`
 - `elements`: Lista degli elementi contenuti nel Va
`elements ('a,'b)relazione → ('a * 'b)list`
(vedi API `Relazione` in file `Relazione.ml` allegato 16/5/2017)

⁵Va indica un valore astratto

⁶ g^n indica n composizioni di g