Programmazione Funzionale: ADT in OCaml

Sommario: 5 Maggio, 2015

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

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 `P` un programma ed `E` un suo termine (espressione). Se la computazione di `P` valuta `E` al valore primitivo $\mathtt{V_E}$, allora il programma $\mathtt{P}[\mathtt{E} \leftarrow \mathtt{V_E}]$ 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

[1] intesa come stato

[2] $\mathtt{P}[\mathtt{E} \leftarrow \mathtt{V_E}]$ indica il rimpiazzamento di ogni occorrenza di `E` in `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:
    - calcolati da un'applicazione di funzione
    - passati 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 luci relativi)

[3] ad es. interi di Barendregt, vedi lucidi su Lambsa-Calcolo e Combinatori
[4] ad es. 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 valutazione
  - un'espressione da valutare in tale ambiente
- **Funzioni**:
  - sono definite da espressioni:
    Ad es.: (OCaml) `fun x` $\rightarrow$ `let...in exp`
  - Possono contenere blocchi-espressione
  - 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
  - **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 sei 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);;
```