Integrazione alle dispense: alberi n-ari

Marta Cialdea Mayer

1 Introduzione

In questa integrazione alle dispense esaminiamo tre modi fondamentali di lavorare sugli alberi n-ari. I primi due (mediante l'uso di funzioni di ordine superiore e mediante la mutua ricorsione) sono già presentati nelle dispense (e qui viene ripresentato sinteticamente, senza spiegazioni). Il terzo modo, presentato nel capitolo sui moduli del libro *Introduzione alla Programmazione Funzionale* di M. Cialdea Mayer e C. Limongelli (Esculapio 2002) – e riportato, sul sito del corso `http://cialdea.dia.uniroma3.it/teaching/pf/materiale/slides/PDF/Cap5-Moduli.pdf` – non è invece stato riportato sulle dispense e lo presentiamo qui, anche indipendentemente dall'uso dei moduli.

2 Rappresentazione di alberi n-ari

Per la rappresentazione di alberi n-ari in OCaml, utilizzeremo la struttura dati così definita:

```
type 'a ntree = Tr of 'a * 'a ntree list
```

e definiamo una funzione di utilità per costruire alberi costituiti da un unico nodo:

```
(* leaf: 'a -> 'a ntree *)
(* leaf x = foglia etichettata da x *)
let leaf x = Tr(x,[])
```

Ad esempio, l'albero di interi `t` rappresentato nella figura 1 è rappresentato dal valore della variabile `t` così definita (utilizzeremo a volte questo albero negli esempi):

```
let t = Tr(1,[Tr(2,[Tr(3,[leaf 4;
                          leaf 5]);
                    Tr(6,[leaf 7]);
                    leaf 8]);
              leaf 9;
              Tr(10,[Tr(11,[leaf 12;
                            leaf 13;
                            leaf 14]);
                     leaf 15;
                     Tr(16,[leaf 17;
                            Tr(18,[leaf 19;
                                   leaf 20])])])])
```

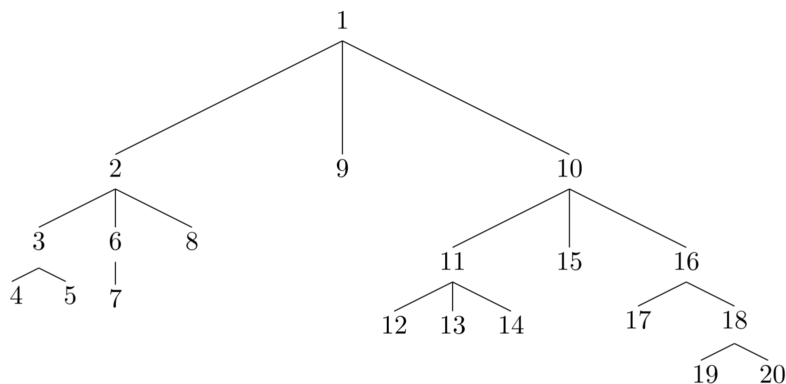

Figura 1: Un albero di esempio

3 Alberi n-ari trattati in modo astratto

Quando si lavora sugli alberi utilizzando funzioni di ordine superiore sulle liste o mediante la mutua ricorsione, gli alberi vengono trattati come tipi concreti, cioè il modo in cui vengono rappresentati (radice e lista di sottoalberi) è accessibile.

Quando invece non si vuole manipolare direttamente la struttura concreta che implementa gli alberi n-ari, si devono definire un insieme di operazioni di base su di essa in modo che gli alberi possano essere trattati come tipo astratto. Questo è particolarmente importante quando si vuole nascondere la rappresentazione concreta degli alberi, ad esempio definendo un modulo apposito la cui interfaccia nasconde il tipo concreto utilizzato per la rappresentazione.

In altri termini, si devono identificare e definire tutti i costruttori, i selettori e i predicati necessari per lavorare sugli alberi. Questa caratterizzazione è analoga a

quella di determinare le operazioni di base sulle liste: i costruttori "lista vuota" e "cons", i selettori "testa" e "resto" della lista, e il "test lista vuota".

Le operazioni di base sugli alberi n-ari, che ne consentono la rappresentazione come tipo astratto di dati sono le seguenti:

- `leaf: 'a ->'a ntree`; il valore di `leaf x` è l'albero n-ario costituito da un unico nodo etichettato da `x` (la stessa già utilizzata come funzione di utilità).
- `join: 'a ntree -> 'a ntree -> 'a ntree`; il valore di `join t1 t2` è l'albero che si ottiene aggiungendo a `t1` un nuovo sottoalbero, `t2`, come suo primo sottoalbero (più a sinistra).
- `root: 'a ntree -> 'a`; il valore di `root t` è la radice di `t` (si noti che tutti gli alberi hanno una radice).
- `left: 'a ntree -> 'a ntree`; il valore di `left t` è il primo sottoalbero di `t`, se esso esiste, altrimenti dà errore.
- `tree_rest: 'a ntree -> 'a ntree`; il valore di `tree_rest t` è l'albero che si ottiene da `t` eliminando il suo primo sottoalbero, se questo esiste, altrimenti dà errore.
- `is_leaf: 'a ntree -> bool` è un predicato; `is_leaf t` restituisce `true` se l'albero `t` è costituito da un unico nodo, `false` altrimenti.

Le prime due funzioni (`leaf` e `join`) sono i costruttori, mentre `root`, `left` e `tree_rest` sono i selettori.

Queste operazioni principali si possono implementare come segue, utilizzando il tipo `'a ntree`:

```
exception Error

(* leaf : 'a -> 'a ntree *)
let leaf a = Tr(a,[])

(* join : 'a ntree -> 'a ntree -> 'a ntree *)
let join (Tr(x,tlist)) t1 =
  Tr(x,t1::tlist)

(* root : 'a ntree -> 'a *)
let root (Tr(x,_)) = x

(* left : 'a ntree -> 'a ntree *)
let left = function
    (Tr(_,t1::_)) -> t1
  | _ -> raise Error

(* tree_rest : 'a ntree -> 'a ntree *)
```

```
let tree_rest = function
    Tr(x,_::tlist) -> Tr(x,tlist)
  | _ -> raise Error

(* is_leaf : 'a ntree -> bool *)
let is_leaf (Tr(_,tlist)) =
  tlist = []
```

Nel resto di questo documento rivediamo alcune funzioni già definite utilizzando funzioni di ordine superiore e/o la mutua ricorsione, e risolviamo gli stessi problemi anche utilizzando questa rappresentazione astratta.

4 Definizione di alcune funzioni su alberi n-ari utilizzando i tre diversi approcci

4.1 Dimensione di un albero (numero di nodi)

- Con la mutua ricorsione:

```
(* size : 'a ntree -> int
   size t = numero di nodi di t *)
(* sumofsizes: 'a ntree list -> int
    sumofsizes [t1;...;tn] = size t1 + .... + size tn *)
let rec size (Tr(_,tlist)) =
  1 + sumofsizes tlist
and sumofsizes = function
    [] -> 0
  | t::rest -> (size t) + sumofsizes rest
```

- Usando List.map:

```
(* sumof: int list -> int *)
let sumof = List.fold_left (+) 0

let rec size (Tr(_,tlist)) =
  1 + sumof (List.map size tlist)
```

- Con l'uso dei selettori:

```
let rec size t =
  if is_leaf t then 1
  else (size (left t)) + size (tree_rest t)
```

Ovviamente quest'ultimo modo di operare sugli alberi si puo' utilizzare anche trattandoli in modo concreto:

```
let rec size (Tr(x,tlist)) =
 match tlist with
  [] -> 1
 | t::rest -> size t + size (Tr(x,rest))
```

4.2 Visita in preordine

- Con la mutua ricorsione:

```
(* preorder : 'a ntree -> 'a list
   preorder t = lista dei nodi di t nell'ordine in cui sarebbero
   visitati secondo la visita in preordine *)
(* preorder_tlist : 'a ntree list -> 'a list
   preorder_tlist [t1;...;tn] = (preorder t1) @ ... @ (preorder tn) *)

let rec preorder (Tr(x,tlist)) =
    x::preorder_tlist tlist
and preorder_tlist = function
    [] -> []
  | t::ts -> preorder t @ preorder_tlist ts
```

- Con List.map e List.flatten:

```
let rec preorder (Tr(x,tlist)) =
    x::List.flatten (List.map preorder tlist)
```

oppure con List.concat_map (solo dalla versione 4.10.0 di OCaml):

```
let rec preorder (Tr(x,tlist)) =
    x::List.concat_map preorder tlist
```

- Mediante l'uso dei selettori:

```
let rec preorder t =
  root t ::
  (if is_leaf t then []
  else preorder (left t) @ List.tl (preorder (tree_rest t)))
```

Si noti che con l'uso dei selettori la visita dell'albero avviene in realtà in postordine. Per questo è necessario togliere il primo elemento del valore di `preorder (tree_rest t)` (la radice viene aggiunta in testa a tutta la lista).

Questo stesso approccio si può adottare anche trattando gli alberi in modo concreto:

```
let rec preorder (Tr(x,tlist)) =
  x ::
  match tlist with
    [] -> []
  | t::rest -> preorder t @ List.tl (preorder (Tr(x,rest)))
```

4.3 Altezza di un albero

- Con la mutua ricorsione:

```
(* height : 'a ntree -> int
   height t = altezza di t *)
(* hl: 'a ntree list -> int
    hl tlist = se tlist <>[] allora il massimo tra le
               altezze degli alberi in tlist,
                  altrimenti errore *)
let rec height (Tr(x,tlist)) =
  match tlist with
    [] -> 0
  | _ -> 1+ hl tlist
and hl = function
    [] -> failwith "h"
  | [t] -> height t
  | t::rest -> max (height t) (hl rest)
```

- Con List.map:

```
(* maxl: 'a list -> 'a
   maxl lst = elemento massimo di lst *)
let rec maxl = function
    [x] -> x
  | x::xs -> max x (maxl xs)
  | _ -> failwith "maxl"

let rec height (Tr(x,tlist)) =
  match tlist with
    [] -> 0
  | _ ->  1 + maxl (List.map height tlist)
```

- Con l'uso dei selettori:

```
let rec height t =
  if is_leaf t then 0
  else max (1+height (left t)) (height (tree_rest t))
```

Stesso approccio, ma usando il tipo concreto:

```
let rec height (Tr(x,tlist)) =
  match tlist with
    [] -> 0
  | t::rest ->
      max (1+height t) (height (Tr(x,rest)))
```

4.4 Fattore di ramificazione

- Con la mutua ricorsione:

```
(* branching_factor :  'a ntree -> int *)
(* blist: 'a tree list -> int
    blist tlist = massimo tra i fattori di ramificazione
                  degli alberi in tlist *)
let rec branching_factor (Tr(_,tlist)) =
  match tlist with
    [] -> 0
   | _ -> max (List.length tlist) (blist tlist)
and blist = function
    [] -> failwith "impossibile"
  | [t] -> branching_factor t
  | t::rest ->
      max (branching_factor t) (blist rest)
```

- Con List.map:

```
let rec branching_factor (Tr(_,tlist)) =
  match tlist with
    [] -> 0
   | _ -> max (List.length tlist)
              (maxl (List.map branching_factor tlist))
```

- Con uso dei selettori. È necessario definire una funzione (`subtrees`) che riporta il numero di sottoalberi di un albero:

```
(* subtrees: 'a ntree -> int *)
let rec subtrees t =
  if is_leaf t then 0
  else 1 + subtrees (tree_rest t)

let rec branching_factor t =
  if is_leaf t then 0
  else
    max (subtrees t)
      (max (branching_factor (left t))
         (branching_factor (tree_rest t)))
```

Stesso approccio, ma usando il tipo concreto:

```
let rec branching_factor (Tr(x,tlist)) =
  match tlist with
    [] -> 0
  | t::rest ->
      max (List.length tlist)
(max (branching_factor t)
           (branching_factor (Tr(x,rest))))
```

4.5 Test di esistenza di un nodo in un albero

- Con la mutua ricorsione:

```
(* occurs_in : 'a ntree -> 'a -> bool
     occurs_in t x = true se x occorre in t *)
(* occurs_in_tlist : 'a ntree list -> 'a -> bool
     occurs_in_tlist tlist x = true se x occorre in almeno uno degli
                               alberi in tlist *)
let rec occurs_in (Tr(x,tlist)) y =
    x=y || occurs_in_tlist tlist y
and occurs_in_tlist tlist y =
  match tlist with
    [] -> false
  | t::ts -> occurs_in t y || occurs_in_tlist ts y
```

- Con List.exists:

```
let rec occurs_in (Tr(x,tlist)) y =
    x=y || List.exists (fun t -> occurs_in t y) tlist
```

- Con i selettori:

```
let rec occurs_in t y =
  y = root t ||
  not (is_leaf t) && (occurs_in (left t) y || occurs_in (tree_rest t) y)
```

Stesso approccio, ma con tipo concreto:

```
let rec occurs_in (Tr(x,tlist)) y =
  y=x ||
   match tlist with
     [] -> false
   | t::rest ->
       occurs_in t y || occurs_in (Tr(x,rest)) y
```

4.6 Ricerca di un ramo mediante backtracking

Ricerca di un ramo dalla radice dell'albero a una *foglia* etichettata da un valore dato.

- Con la mutua ricorsione:

```
exception NotFound;;

(* path : 'a ntree -> 'a -> 'a list
     path t y = cammino fino a una foglia etichettata da y in t
   path_tlist : 'a ntree list -> 'a -> 'a list
     path_tlist tlist y = cammino fino a una foglia etichettata da y in
                          uno degli alberi in tlist *)
let rec path (Tr(x,tlist)) y =
  match tlist with
    [] ->
      if x=y then [x]
      else raise NotFound
  | _ -> x::path_tlist tlist y
and path_tlist tlist y = match tlist with
  [] -> raise NotFound
| t::ts -> try path t y
          with NotFound -> path_tlist ts y
```

- In questo caso non è ragionevole utilizzare funzioni di ordine superiore: si potrebbe anche fare, ma sarebbe computazionalmente molto più costoso.

- Con i selettori:

```
let rec path t y =
  if is_leaf t
  then
    if y = root t then [y]
    else  raise NotFound
  else
    try root t :: path (left t) y
    with _ ->
      let rest = tree_rest t in
      if is_leaf rest then raise NotFound
      else path rest y
```

  Si noti la necessità di considerare a parte il caso di albero con un solo sottoalbero, cioè quando `tree_rest t` è una foglia (albero senza sottoalberi). Infatti, in questo caso non sarebbe corretto riportare il cammino in `tree_rest t`: se abbiamo un albero con n sottoalberi, la sua radice non è una foglia, ma le varie chiamate ricorsive, togliendogli via via i sottoalberi, arriveranno a considerare

l'albero con nessun sottoalbero. Ma questo non deve essere considerato come foglia.

Utilizzando il tipo concreto, il caso “albero con un unico sottoalbero” sarà trattato mediante il pattern matching:

```
let rec path (Tr(x,tlist)) y =
  match tlist with
    [] ->
      if y = x then [y]
      else  raise NotFound
  | [t] ->  x::path t y
  | t::rest ->
      try x::path t y
      with _ -> path  (Tr(x,rest)) y
```