Principles and Paradigms of Programming Languages

a.a. 2017/18

Esercizi Haskell (2)

9 novembre 2017

Esercizi su tipi user-defined

- Dati i tipi delle funzioni parziali e delle tabelle (liste di associazioni da chiavi a valori)

```
type PFun a b  = a -> Maybe b
type Table k v = [(k,v)]
```

  definire, indicando anche la type signature:

  `makeTotal f def_b` la funzione totale ottenuta "completando" la funzione parziale `f` con il valore di default `def_b` dove non è definita
  `consFun ps` la funzione parziale che data una tabella `ps = [(a_1,b_1), ... (a_n,b_n)]` vale `b_i` su `a_i`, è indefinita altrimenti[1].

- Dato il tipo degli alberi binari (`deriving Show` serve a renderli "visualizzabili"), definire

```
data BTree a = Empty | Node  a (BTree a) (BTree a) deriving Show
```

  `frontier t` la frontiera di un albero (lista delle etichette delle foglie)
  `inorder f a t` esegue la visita inorder dell'albero con parametro di accumulazione `a`, a ogni nodo con etichetta `b` il nuovo valore del parametro di accumulazione è `f a b`
  `inorder_list` (istanza della precedente) la lista delle etichette dei nodi con visita inorder
  `sum_tree` (istanza della precedente) la somma dei nodi di un albero con etichette numeriche
  `node_num` (istanza della precedente) il numero dei nodi

Interprete per il linguaggio $\mathcal{E}$ Si implementi il calcolo $\mathcal{E}$ visto a lezione. In particolare:

- Si definisca un tipo, sia `Exp`, corrispondente ai termini del linguaggio (attenzione a non andare in conflitto con costruttori predefiniti come `True` e `False`).
- Si definisca una funzione `isNum` che controlla se un termine è un numerale, ossia della forma $n ::= 0 \mid \texttt{succ}\ n$.
- Si definisca una funzione `isVal` che controlla se un termine è un valore.
- Si definisca una funzione `reduce :: Exp -> Maybe Exp` che corrisponde alla relazione di riduzione $\rightarrow$, quindi restituisce `Nothing` se e solo se l'argomento è una forma normale.
- Si definisca una funzione `reduceStar :: Exp -> Exp` che corrisponde alla relazione $\rightarrow^\star$, quindi se l'argomento è una forma normale restituisce l'argomento stesso.
- Esercizio aggiuntivo per chi va veloce: si definisca una funzione `bigstep :: Exp -> Maybe Exp` che corrisponde alla relazione di riduzione big-step $\Downarrow$. Attenzione: il risultato deve essere della forma `Just v` con `v` valore, oppure `Nothing` se l'argomento non si valuta a un valore.

[1] Assumiamo che un'associazione per la stessa chiave sovrascriva la precedente.

Esercizi su lazy evaluation

- Definire una funzione `prime` che restituisce l'i-esimo numero primo, e una funzione che calcola il primo numero primo maggiore di un certo numero.
- Definire delle funzioni analoghe a `repeat`, `take` e `replicate` viste a lezione per il tipo di dato degli alberi binari dato sopra.
- Definire un operatore `(==>) :: Bool -> Bool -> Bool` corrispondente all'implicazione usuale non stretta (o *corto-circuitata*), ossia tale che `False ==> x` si valuti in `True` senza dover valutare `x`.
- Definire una funzione `implies` come la precedente, ma stretta, ossia tale che `False ==> x` diverga se la valutazione di `x` diverge.
- Definire un'ulteriore variante che non diverge quando il secondo argomento è `True`. Come possiamo interpretare intuitivamente questa versione? È possibile combinarla con la prima, ossia dare una versione che non diverge quando il primo argomento è `False` o il secondo è `True`?
- Definire una funzione `dupli` che data una lista `[x_1, ..., x_n]` restituisce la lista `[x_1, x_1, ..., x_n, x_n]` utilizzando: list comprehension, `foldl`, `foldr`. Si esamini il comportamento delle diverse versioni nel caso di liste infinite e lo si spieghi.

Difference lists Date due liste `xs` ed `ys`, la valutazione di `xs ++ ys` è lineare nella lunghezza di `xs`. Nei casi in cui si costruiscono liste aggiungendo frequentemente nuovi elementi in fondo (per esempio, manipolando stringhe), occorre una struttura dati diversa che consenta di concatenare sia in testa che in coda in tempo costante. Si può utilizzare a questo scopo il seguente tipo[2] delle *difference list*, in cui una lista `xs` è rappresentata da una funzione che concatena `xs` in testa a una lista presa in input[3]:

```
type DList a = [a] -> [a]
```

Dato questo tipo, definire le funzioni che seguono (dove n è la lunghezza della lista).

- `fromList :: [a] -> DList a`, che presa una lista la trasforma in una difference list in O(1).
- `toList :: DList a -> [a]`, che riconverte una difference list in una normale lista in O(n).
- `append :: DList a -> DList a -> DList a`, che concatena due difference list in O(1).
- `empty :: DList a`, la difference list vuota.
- `cons :: a -> DList a -> DList a`, che aggiunge in testa un elemento a una difference list dello stesso tipo in O(1) (come `(:)` per le liste).

[2] Alternativamente, è possibile dare la seguente definizione: `data DList a = DList ([a] -> [a])`, che, se data internamente a un *modulo*, permette di nascondere l'implementazione non esportando i data constructor.

[3] Esempio tratto dal capitolo 13 di Real World Haskell.