Principles and Paradigms of Programming Languages

a.a. 2017/18

Esercizi Haskell (3)

27 novembre 2017

Sistema di tipi per il linguaggio $\mathcal{E}$
Si implementi il sistema di tipi per $\mathcal{E}$ visto a lezione. In particolare:

- Si definisca un tipo, sia `Type`, corrispondente ai tipi del linguaggio.
- Si definisca una funzione `typeof` che restituisce il tipo di un termine. Tale funzione può avere semplicemente tipo `Exp -> Maybe Type`, modellando quindi l'errore con `Nothing`, oppure tipo `Exp -> Either TypeError Type` con `TypeError` un opportuno tipo che modelli i diversi errori di tipo possibili.

Esercizi su input/output

- Si scriva una funzione `doGuessing :: Int -> IO()` che richiede ripetutamente di indovinare un numero dato, rispondendo a seconda dei casi `Too high!`, `Too low!`, `You win!`, finché non si indovina.
- Si scriva una funzione `mysequence :: [IO a] -> IO [a]` che prende in input una lista di azioni di input/output e le esegue in sequenza, collezionandone i risultati in un'unica lista.

Esercizi su funtori

- Si renda istanza di `Functor` il costruttore di tipo `BTree` definito precedentemente.
- Si modifichi la definizione di `Table` vista a lezione in modo da poterla rendere un funtore (`fmap` dovrà applicare una funzione `v -> v'` a una tavola di tipo `Table k v` restituendo una tavola di tipo `Table k v'`.)

Lambda calcolo Si implementi il lambda calcolo call-by-value visto a lezione, da solo o estendendo la precedente implementazione del linguaggio $\mathcal{E}$. Si modellino per semplicità le variabili come numeri interi, assumendo convenzionalmente che in un termine da ridurre le variabili libere siano numeri negativi, quelle legate siano numeri da 0 in poi. In particolare:

- Si definisca un tipo, sia `Exp`, corrispondente ai termini del linguaggio.
- Si definisca una funzione `freeVars :: Exp -> [Var]` che restituisce l'insieme $fv(t)$ delle variabili libere di un termine t. Per l'implementazione, si vedano le funzioni `union` e `(\\)` del modulo `Data.List`.
- Si definisca una funzione `allVars :: Exp -> [Var]` che restituisce l'insieme $v(t)$ di tutte le variabili di un termine t.
- Si definisca una funzione `subst` che implementa la sostituzione $t[t'/x]$ secondo la seguente versione più concreta della definizione vista a lezione, che esplicita la ridenominazione:
  - $x[t/x] = t$
  - $y[t/x] = y$ se $x \neq y$
  - $(\lambda x.t)[t/x] = \lambda x.t$
  - $(\lambda y.t_1)[t_2/x] = \lambda y.(t_1[t_2/x])$ se $x \neq y$, $y \notin fv(t_2)$
  - $(\lambda y.t_1)[t_2/x] = (\lambda y'.t_1[y'/y])[t_2/x]$ se $x \neq y$, $y \in fv(t_2)$, con $y' \notin fv(t_2) \cup v(\lambda y.t_1)$
    (si noti che grazie alla convenzione scelta y' può essere semplicemente ottenuta come la prima variabile legata che non appartiene all'insieme indicato)
  - $(t_1\ t_2)[t/x] = t_1[t/x]\ t_2[t/x]$

- 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 $\rightarrow$.
- Si definisca una funzione `reduceStar :: Exp -> Exp` che corrisponde alla relazione $\rightarrow^\star$.
- Si dia un'opportuna dichiarazione di `Exp` come istanza di `Show`.

Monade di parser Possiamo modellare in modo semplice un parser come una funzione `String -> [(a, String)]` che, data una stringa in input, restituisce una lista di coppie: elemento di un certo tipo `a` risultato del parsing e parte "non consumata" della stringa[1] . In particolare, modelliamo con la lista vuota il fallimento del parsing, mentre consideriamo una lista di coppie per modellare anche situazioni in cui una stringa può essere "parsata" in più modi, anche se considereremo solo casi deterministici.

1. A partire da questa idea si definiscano:
   - `parse :: Parser a -> String -> [(a, String)]` la funzione che applica un parser a una stringa
   - la monade `Parser`, in cui `return a` sarà il parser che restituisce sempre l'elemento `a`, e `>>=` la composizione sequenziale di parser
   - `failure :: Parser a` il parser che fallisce sempre
   - `item :: Parser Char` il parser che legge il primo carattere della stringa in input, fallisce se è vuota.
   - `+++ :: Parser a -> Parser a -> Parser a` il parser che prima prova ad applicare il primo parser alla stringa in input, e se questo fallisce applica il secondo.

   Avendo dato queste definizioni, e per esempio la seguente funzione:

   ```
   myparser :: Parser (Char, Char)
   myparser = do
     c1 <- item
     item
     c2 <- item
     return (c1, c2)
   ```

   si dovrà ottenere:

   ```
   > parse (failure+++myparser) "abcd"
   [(('a','c'),"d")]
   >parse (myparser+++return ('h','g')) "ab"
   [(('h','g'),"ab")]
   ```

2. Utilizzando le precedenti definizioni, si definiscano:
   - `sat p :: (Char -> Bool) -> Parser Char` il parser che legge un singolo carattere che soddisfa `p`, altrimenti fallisce
   - utilizzando il precedente e appropriati predicati di `Data.Char`, i parser `digit`, `upper`, `lower`, `letter`, e `char :: Char -> Parser Char` che legge un certo carattere
   - `string :: String -> Parser String` che legge una certa stringa
   - `many p, many1 :: Parser a -> Parser [a]` che applicano un parser quante volte possibile prima di fallire, restituendo la lista dei risultati (`many` accetta anche zero applicazioni, mentre `many1` ne vuole almeno una).
   - `space :: Parser ()` che salta tutti gli spazi, ossia i caratteri per cui vale `isSpace`
   - `symbol :: String -> Parser String` che legge una certa stringa saltando tutti gli spazi prima e dopo.

Parser per $\mathcal{E}$ Utilizzando le precedenti definizioni, si scriva un semplice parser `term :: Parser Exp` per il linguaggio $\mathcal{E}$ implementato precedentemente.

Uso di `Maybe` e `Either` come monadi Si rivedano gli esercizi precedenti utilizzando, dove conveniente, le operazioni monadiche e la do notation. In particolare risulta possibile definire in modo più compatto la riduzione `reduce :: Exp -> Maybe Exp` e il typechecking `typeof :: Exp -> Either TypeError Type`.

[1] La libreria Haskell fornisce parser più sofisticati, si veda il package `Parsec`.