Linguaggi di Programmazione

Ivano Salvo

FP8: Definizione di tipi in Haskell

Corso di Laurea in **Matematica**

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **10**, 1 dicembre 2020

Dalla Lezione 9

Distruttori

Deallocazione di Oggetti

Gli stessi problemi sorgono quando e come deallocare gli oggetti.

Ci sono **deallocazioni automatiche dovute a oggetti allocati sullo stack**: quando usciamo da un contesto (ad esempio, una funzione finisce di eseguire) va deallocata tutta la memoria che appartiene a quel contesto.

Ma nel caso della nostra classe HowMany? come garantire la correttezza del conteggio quando vengono deallocati gli oggetti?

Inoltre: la deallocazione di una struttura dati o di un oggetto aggregato deve causare la deallocazione degli oggetti riferiti? Risposta difficile, ma come farla?

Ovviamente, anche in questo caso, il C++ genera per ogni classe un **distruttore di default**.

Distruttore

Per programmare opportunamente cosa accade alla deallocazione di un oggetto **è necessario definire** un codice più significativo per **il distruttore** che ha sempre prototipo: `~C()` dove `C` è il nome della classe.

Il **distruttore** viene chiamato **automaticamente** sugli oggetti allocati sullo **stack**, **ogni volta che si esce dal contesto** a cui appartengono (tipicamente una funzione o un blocco -- **scope**).

Deve invece essere chiamato **esplicitamente** per gli oggetti **allocati sull'heap** per effetto di una `new`. Si fa usando il comando `delete ref;` dove ref è un puntatore all'oggetto da eliminare.

```
public:
      /* distruttore */
   ~HowMany(){objectCount--;}
   ...
```

Quando e cosa deallocare?

La **deallocazione** di memoria è una delle più **fertili sorgenti di errori** nella programmazione.

Anche se l'uso di `new` e `delete` **migliora la situazione rispetto alle primitive C** `malloc/calloc` e `free` ci sono questioni spesso difficili da dirimere mentre si programma.

Ad esempio: se dealloco la struttura dati che mi è servita a contenere i pezzi per valutare lo scacco al re, devo deallocare anche i pezzi? Qui la risposta è ovviamente No, ma le cose potrebbero non essere sempre chiare.

È viceversa chiaro che se deallocassi la scacchiera, dovrei voler deallocare i pezzi riferiti dalla scacchiera e allocati all'inizio della partita.

Occorre sempre chiedersi (in fase di progettazione) **a chi appartengono gli oggetti** e di chi è la **responsabilità di crearli e distruggerli**.

Lezione 10a

Definizioni di tipi di Dato in Haskell

Dichiarazioni di sinonimi di tipo

Con la parola chiave `type` possiamo definire **nuovi nomi di tipo**.

Si tratta essenzialmente di sinonimi, che tuttavia possono essere utili per introdurre un livello di astrazione. Tuttavia non offrono nessun livello di protezione sui dati.

```
 -- Posso definire un tipo Casella e Mossa
type Casella = (Int, Int)
type Move = (Int, Int)
 -- Posso dichiarare il tipo di una funzione move:
move :: Casella -> Move -> Casella
move (x,y)(dx,dy)=(x+dx, y+dy)
 -- ma anche:
move' (x,y)(dx,dy)=(x+dx, y+dy)
>t move'
move'::(Num t1,Num t) => (t, t1) -> (t, t1) -> (t, t1)
 -- Casella, Posizione sono compatibili
move'' p p' = move' (move p p') (1,1)
>t move''
move':: Casella -> Move -> (Int, Int)
```

Sinonimi parametrici

I sinonimi possono essere anche **parametrici**, cioè possono **dipendere da variabili di tipo**. Vediamo 2 simpatici esempi e i controlli fatti dal compilatore Haskell:

```
 -- Posso rinominare il tipo coppia
type Pair a b = (a, b)
 -- oppure le coppie omogenee
type PairH a = (a, a)
maxP:: Ord a => PairH a -> a -- definisco il tipo
maxP (x, y) = if x > y then x else y
fstP:: PairH a -> a
fstP (x, y) = x
fstQ:: Pair a b -> a
fstQ (x, y) = x
 -- qualche controllo viene fatto, però:
>fstP (2, [2])
 -- dà errore, perché non è una PairH, ma...:
>fstQ (2, [2])
2
```

Dichiarazioni di nuovi tipi

Più interessante la definizione di nuovi tipi attraverso l'**uso di costruttori**. Ci sono numerosi, famosi, tipi finiti, ad esempio il tipo `Bool` o il tipo dei `SetteNani`.

Anche questi tipi possono dipendere da variabili di tipo. Ecco il famoso tipo `Maybe` che è il tipo di **computazioni che possono non andare a buon fine**.

```
 -- Booleani
data Bool = False | True
 -- Sette Nani
data SetteNani = Eolo | Pisolo | Brontolo | …
 -- Eccezioni: tipi parametrici
data Maybe a = Nothing | Just a
 -- Definizioni di funzioni di tipo Maybe
safeHead [] = Nothing
safeHead (x:xs) = Just x
>:t safeHead
safeHead :: [a] -> Maybe a
```

Tipi Induttivi o Ricorsivi

In Haskell si possono facilmente definire tipi **induttivi** o **ricorsivi** esattamente come visto finora, semplicemente **specificando la segnatura** (o tipo) **dei costruttori**.

Questo modo di procedere generalizza il tipo delle liste predefinite, costruite a partire da **lista vuota** `[]` e dalla funzione **cons** `(:)`.

```
 -- Naturali "unari"
data Nat = Zero | Succ Nat
 -- Liste
data List a = Nil | Cons a (List a)
 -- Alberi binari
data BTree a = Leaf a | Node (BTree a) (BTree a)
 -- o anche:
data BTree'a = Empty | Node (Btree' a) (Btree' a)
 -- anche:
data Lambda = Var Int | Apply Lambda Lambda |
     Abs Int Lambda
kappa = Abs 1 (Abs 2 (Var 1))
ide = Abs 6 (Var 6)
```

Definizioni di funzioni

Le funzioni si possono sempre definire **per induzione** sui costrutturori, usando il **pattern matching**, esattamente come per le liste. Significa che ho gratis, i **distruttori**.

Vediamo la lunghezza sulle liste.

L'append tra liste.

E la visita in order che produce una lista.

```
 -- Lunghezza di una lista
length Nil = Zero
length (Cons _ l) = Succ (length l)
 -- append tra liste
append l Nil = l
append Nil l = l
append (Cons x l) m = Cons x (append l m)
(Cons 1 (Cons 2 Nil)) (:) []
-- visita inorder di un albero binario
flatten (Leaf x) = Cons x Nil
flatten (Node x t1 t2) =
   append (flatten t1) (Cons x (flatten t2))
```

Lezione 10b

Definizioni di Classi in Haskell

Classi e subtyping

Abbiamo visto, fin dalla prima lezione tipi a volte misteriosi.

Facciamo l'esempio della merge.

Se chiedo il tipo di `merge`, ottengo prima del tipo la scritta "`Ord a =>`" che significa grossomodo "**per ogni tipo a che su cui è definita una relazione d'ordine**" o meglio ancora "**per ogni sottotipo a della classe `Ord`**".

Questo perché merge non è completamente polimorfa rispetto al tipo degli elementi della lista in quanto richiede di valutare <.

```
 -- fusione ordinata di liste ordinate
merge xs [] = xs
merge [] xs = xs
merge xs@(x:txs) ys@(y:tys)
  |   x < y   = x:merge txs ys
  | otherwise = y:merge xs tys

>:t merge
merge :: Ord a => [a] -> [a] -> [a]
```

Classi predefinite (1)

Le **classi** possono essere viste in Haskell come una **collezione di tipi**, un po' come i tipi sono una collezione di valori.

Vediamo le classi predefinite in Haskell.

Equality Types, `Eq`: pretende l'esistenza di due funzioni binarie

```
(==): a->a->Bool
(/=): a->a->Bool
```

Tutti i tipi base (`Bool`, `Char`, `String`, `Int`, `Integer`, `Float` e `Double`) sono istanze di `Eq`. Liste e tuple sono anche equality types quando contengono elementi di equality types.

Ordered Types, `Ord`: sono **equality types** che hanno inoltre le seguenti funzioni di ordinamento:

```
(<)(<=)(>)(>=): a->a->Bool
min, max: a->a->a
```

Come prima, tutti i tipi base appartengono a Ord e anche liste e tuple lo sono se contengono elementi di un tipo ordinato (liste e tuple sono ordinate secondo l'ordine lessicografico).

[1,5,6,7]<[2], [1]<[1,2]

Classi predefinite (2)

Numeric Types, `Num`: pretende l'esistenza delle seguenti funzioni:

```
(+),(-),(*): a->a->a
negate, abs, signum: a->a
```

Osservate che ad esempio la costante 3 in Haskell ha tipo `(Num a)=>a`, perché può appartenere tanto a `Int`, `Integer`, `Float` o `Double`.

Integral Types, `Integral`: sono **numeric types** che hanno inoltre le seguenti funzioni:

```
div, mod: a->a->a
```

Fractional Types, `Fractional`: sono **numeric types** che hanno inoltre le seguenti funzioni:

```
recip, (/): a->a->a
```

Sono fractional types per esempio `Float` e `Double`.

Classi predefinite (3)

Showable Types, `Show`: pretende l'esistenza della funzione:

```
show: a->String
```

Readable Types, `Read`: sono **numeric types** che hanno inoltre le seguenti funzioni:

```
read: String->a
```

Definizioni di Classi

È ovviamente possibile **definire nuove classi**, così come è possibile definire nuovi tipi.

Vediamo la definizione della classe `Eq`: occorre **specificare le operazioni** e i **relativi tipi**.

È possibile anche scrivere del codice significativo: per esempio dire che /= è la negazione di == come ci si aspetta da una qualsiasi relazione di uguaglianza.

```
 -- definizione della classe Eq
class Eq a where
   (==), (/=) :: a -> a -> Bool

   x /= y = not (x == y)
```

Definizioni di Istanze

Dopodiché è necessario avere un meccanismo per dire che un certo tipo **è istanza di una classe**.

Ad esempio, il tipo booleano ammette uguaglianza e può dunque essere dichiarato istanza di `Eq`.

Occorre contestualmente fornire il codice di == mentre non è necessario fornire quello di /= in quanto definito in modo standard come la negazione di ==.

Solo i tipi definiti con `data` (e non con `newtype`) possono essere dichiarati istanze

```
 -- definizione di Bool come istanza di Eq
instance Eq Bool where
   False == False = True
   True  == True  = True
     _   ==   _   = False
```

Estensioni di Classi

Le classi possono formare delle gerarchie ed essere **estese**, analogamente alle definizioni di sottoclassi.

Ad esempio, la classe `Ord` estende la classe `Eq`. Vediamo come si definisce.

Anche qui possiamo scrivere codice significativo.

Volendo potremo definire anche <= e >=.

```
 -- definizione di Ord come estensione di Eq
class Eq a => Ord a where
    (<),(<=),(>=),(>) :: a -> a -> Bool
    min, max          :: a -> a -> a

   min x y | x <= y = x | otherwise y
   max x y | x <= y = y | otherwise x
   x <= y = x < y || x==y
   x >= y = x > y || x==y
   -- se totali x > y  = y < x
```

Derivazione da Classi

Ogni volta che si definisce un tipo, è buona norma dichiararlo (o meno) istanza di qualche classe, in particolare delle classi predefinite.

Per le classi `Eq`, `Ord`, `Show` e `Read` è possibile usare un meccanismo automatico (i costruttori sono ordinati nell'ordine in cui vengono scritti dal programmatore)

```
 -- usare deriving per creare istanze
data Bool = False | True
        deriving (Eq, Ord, Show, Read)

> False < True
True
> show False
"False"
```

Esempio: alberi Ennari

A conclusione di questa passeggiata nelle definizioni di tipi e classi vediamo come in Haskell sia relativamente facile definire gli alberi in cui ogni nodo ha un numero arbitrario di figli:

```
 -- Alberi con un numero arbitrario di figli
data Tree a = Empty | [Tree a]
```

Lezione 10c

Input/Output in Haskell

Il problema dell'input/output

Abbiamo usato Haskell semplicemente per valutare funzioni. Scriviamo definizioni di funzioni e valutiamo il valore delle funzioni sugli argomenti.

In realtà, i programmi interattivi sono un problema nel mondo funzionale, a causa del fatto che **input** e **output** sono, a rigore dei **side-effects**: il **valore di una funzione pura dipende solo dai suoi argomenti**.

Idea: immaginare esista un argomento implicito nelle funzioni interattive che rappresenta lo "stato del mondo".

Espressioni di tipo `IO a` sono dette **actions**.

```
 -- prima approssimazione:
type IO = World -> World

 -- ma un programma può anche tornare valori
type IO a = World -> (a, World)
```

Azioni Base

Il tipo `IO Char` è il tipo delle **azioni che tornano un carattere**, mentre `IO ()` è il tipo che torna la tupla vuota come un risultato (= nessun risultato, assomiglia a `void`), cioè il tipo delle azioni che sono solo side-effects.

Osserviamo che il tipo `IO a` è predefinito in Haskell ed è visto come un tipo dalla definizione nascosta.

Ecco le primitive base per:

- leggere un carattere da tastiera,
- scrivere un carattere da tastiera,
- fornire un valore alle azioni (è un ponte tra le funzioni pure e le azioni: è a senso unico, in quanto una volta impuri **non c'è redenzione!**)

```
 -- basic actions:
getChar :: IO Char
putChar :: Char -> IO ()
return :: a -> IO a
```

Sequenze

Usando il costrutto `do` **è possibile mettere in sequenza azioni** in un'unica azione complessa.

L'istruzione return è il modo in cui le azioni possono tornare un risultato ed è un ponte tra il mondo impuro e quello puro: ovviamente **non c'è redenzione** e il contrario non si può fare.

```
 -- sequenza di azioni:
do  v1 <- a1
    v2 <- a2
    ...
    vn <- an
    return (f v1 v2 ... vn)
```

Primitive derivate

```
 -- leggere una stringa
getLine :: IO String
getLine =
   do x <- getChar
      if x == '\n' then
         return []
      else
         do xs <- getLine
         return (x:xs)

 -- scrivere una stringa
putStr :: String -> IO ()
putStr   []    = return ()
putStr (x:xs) = do putChar x
                   putStr xs

 -- scrivere una stringa e andare a capo
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar `\n`
```

Il gioco dell'impiccato (1)

[da *Programming in Haskell*, G. Hutton, 2016]

Come esempio di programma interattivo, vediamo una semplice variazione del gioco dell'impiccato.

```
 -- inizio del gioco:
hangman :: IO ()
hangman = do putStrLen "Think a word: "
             word <- secretGetLine
             putStrLn "Try to guess it: "
             play word

 -- secretGetLine evita di riprodurre la stringa
secretGetLine =
   do x <- secretGetChar
   if x == '\n' then
      do putChar x
         return []
   else
      do putChar '-'
         xs <- secretGetLine
        return (x:xs)
```

Il gioco dell'impiccato (2)

[da *Programming in Haskell*, G.Hutton, 2016]

Occorre interagire col sistema per impedire la riproduzione dei caratteri a video ☺

```
 -- bisogna estendere getChar:
secretGetChar :: IO ()
hangman = do hSetEcho stdin False
             x <- getChar
             hSetEcho stdin True
             return x

 -- ci resta da programmare il ciclo di gioco
play word =
   do putStr "?"
   guess = getLine
   if guess == word then
      putStrLen "You got it!!"
   else
      do putStrLen (match word guess)
         play word
```

Il gioco dell'impiccato (3)

[da *Programming in Haskell*, G.Hutton, 2016]

La funzione match mostra i caratteri comuni tra il tentativo e la parola segreta.

```
 -- funzione che genera la lista risultato:
match xs ys = [if elem x ys then x else `-`
                  | x <- xs]
```

Output del programma:

```
*Main> hangman
Think of a word:
------
Try to guess it:
? s
s-----
? test
se--et
? cross
s-cr--
? secret
You got it!
```

Lezione 10

That's all Folks…

Grazie per l'attenzione…

…Domande?