Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Haskell: Un po' più sui Tipi

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **5**, 15 marzo 2021

Lezione 5a

Classi in Haskell

Classi e subtyping

Finora ho mantenuto il più stretto riserbo su alcuni tipi strani… Ricordiamo l'esempio di `merge`.

Se chiedo il tipo di `merge`, ottengo prima del tipo la scritta "`Ord a =>`" che significa grossomodo "**per ogni tipo `a` 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 e subtyping

Ci sono esempi ancora più banali. Proviamo a chiedere a GHCi il tipo del numero 3.

La costante 3 ha un **tipo polimorfo**, **ma non completamente polimorfo**, perché potrebbe essere un `Integer`, un `Float`, etc.

Similmente, per (+) e altre operazioni aritmetiche definite **su tutti i tipi numerici**.

Leggermente diversa la questione per la divisione `(/)` e per la divisione intera, `div`.

```
 -- tipi numerici strani
> :t 3
3 :: Num a => a
> :t (+)
(+) :: Num a => a -> a -> a
>:t (/)
(/) :: Fractional a => a -> a -> a
>:t div
div :: Integral 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`: comprende tutti i tipi che definiscono 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
```

(**Contro**)**Esempio**: Se provate a valutare una funzione al prompt dell'interprete GHCi, viene visualizzato il seguente messaggio di errore:

```
[Prelude> \x->x

<interactive>:2:1:
    No instance for (Show (t0 -> t0)) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

Una funzione è un valore, ma l'interprete non può stamparla, perché sulle funzioni non è definito il metodo `show` usato poi da `print`.

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

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

Lezione 5b

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 (**equivalenza strutturale**).

```
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)
-- 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 5c

Definizioni di Classi in Haskell

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 5

That's all Folks…

Grazie per l'attenzione…

…Domande?