Introduzione alla programmazione Funzionale – il linguaggio Haskell

Marco Gavanelli

Cos'è un linguaggio funzionale?

Esistono diverse opinioni, ma in generale

- La programmazione funzionale è uno stile di programmazione in cui il metodo base di computazione è l'applicazione di funzioni agli argomenti;
- Un linguaggio funzionale è un linguaggio che supporta ed incoraggia lo stile funzionale;
- Le funzioni vengono interpretate come in matematica

Funzioni nei linguaggi di programmazione

- **Tutti i linguaggi di programmazione supportano un qualche costrutto per esprimere *funzioni* …**
- **…ma solo di rado esse sono *first-class entities***

Cosa significa?

- **Una funzione che sia una "first-class entity" dev'essere *manipolabile come ogni altro tipo di dato*** e quindi:
  - **deve poter essere *assegnata a variabili* (di tipo "funzione"!)**
  - **deve poter essere *passata come argomento a un'altra funzione***
  - **deve poter essere *restituita da un'altra funzione***
  - **deve poter essere *definita e usata "al volo"* come ogni altro valore (literal) di ogni altro tipo**
  - **magari anche senza avere per forza un nome..**

Funzioni nei linguaggi imperativi

- **Nei linguaggi imperativi tradizionali, una funzione tipicamente è solo un *costrutto che incapsula codice*: *non* è manipolabile "*come ogni altro tipo di dato"***
  - **non può essere assegnata a variabili**
    - **approssimazione: i puntatori a funzione del C, che però contengono solo l'indirizzo del codice, non una vera "entità funzione"..**
  - **non può essere passata come argomento a un'altra funzione**
    - **perché gli argomenti devono essere *valori* di un qualche *tipo*, mentre le funzioni in quei linguaggi non lo sono**
  - **non può essere restituita da un'altra funzione**
    - **perché non si può "sintetizzare comportamento" come fosse un valore (e le funzioni esprimono "comportamenti"…)**
    - **al più, si può *restituire l'indirizzo di una funzione già esistente***
  - **non può essere definita e usata "al volo"**
    - **le funzioni devono essere definite nel codice, staticamente**

Funzioni nei linguaggi funzionali

- **Nei linguaggi funzionali, invece, una funzione non è solo un costrutto che incapsula codice: è *un tipo di dato manipolabile come ogni altro***
  - **può essere assegnata a variabili di tipo *"funzione"***
    - **come un valore intero può essere assegnato a una variabile `int`, o una stringa a un oggetto `String`**
  - **può essere passata come argomento a un'altra funzione**
    - **come ogni valore di ogni altro tipo**
  - **può essere restituita da un'altra funzione**
    - **perché sintetizzare un "valore funzione" (ossia, nuovo comportamento), dopo tutto, non è diverso dal sintetizzare un valore di un altro tipo**
    - **una funzione è "solo" un valore.. anche se *eseguibile***
  - **può essere definita e usata "al volo"**
    - **se valori di altri tipi possono essere creati e generati "da programma", perché le funzioni devono essere cablate staticamente nel codice?**

Perché Functional programming?

- La programmazione funzionale è uno dei principali paradigmi di programmazione
- Un nuovo modo di pensare la programmazione
- Stile di programmazione molto usato
  - Microsoft: F#
  - Java: Scala, Clojure
- Idee nate nella programmazione funzionale vengono via-via inserite nei linguaggi "mainstream"
  - Javascript (da sempre ha il costruttore Function e la keyword function per specificare funzioni "anonime"),
  - C#: delegati e costruttore Func<…>
  - C++: dallo standard 11, supporto per funzioni "anonime" e possibilità di passarle
  - Java: supporto *parziale introdotto in Java 8* con funzioni anonime come lambda expressions, si possono definire come literal e passare come argomenti..

Perché Haskell?

- Haskell's expressive power can improve productivity
  - Small language core provides big flexibility
  - Code can be very concise, speeding development
  - Get best of both worlds from compiled and interpreted languages
- Haskell makes code easier to understand and maintain
  - Can dive into complex libraries and understand what the code is doing
- Haskell can increase the robustness of systems
  - Strong typing catches many bugs at compile time
  - Functional code permits better testing methodologies
  - Can parallelize non-concurrent code without changing semantics

9

Alcuni utilizzatori di Haskell

- ABN AMRO
- Alcatel-Lucent
- AT&T
- Bank of America Merril Lynch
- Barclays Capital Quantitative Analytics Group
- Ericsson AB
- Facebook
- Google
- Intel
- Microsoft
- The New York Times
- NICTA
- NVIDIA
- Qualcomm, Inc
- Siemens Convergence Creators GmbH Austria

10

Caratteristiche

Purely functional

Statically typed

Lazy

11

Example

Summing the integers 1 to 10 in Java:

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
  total = total + i;
```

The computation method is variable assignment.

12

Example

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

The computation method is function application.

13

Historical Background

1930s:

Alonzo Church develops the lambda calculus, a simple but powerful theory of functions.

14

Historical Background

1950s:

John McCarthy develops Lisp, the first functional language, with some influences from the lambda calculus, but retaining variable assignments.

15

Historical Background

1960s:

Peter Landin develops ISWIM, the first *pure* functional language, based strongly on the lambda calculus, with no assignments.

16

Historical Background

1970s:

John Backus develops FP, a functional language that emphasizes *higher-order functions* and *reasoning about programs*.

17

Historical Background

1970s:

Robin Milner and others develop ML, the first modern functional language, which introduced *type inference* and *polymorphic types*.

18

Historical Background

1970s - 1980s:

David Turner develops a number of *lazy* functional languages, culminating in the Miranda system.

19

Historical Background

1987:

Haskell
A Purely Functional Language

An international committee of researchers initiates the development of Haskell, a standard lazy functional language.

20

Historical Background

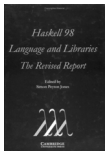
1990s:

Phil Wadler and others develop *type classes* and *monads*, two of the main innovations of Haskell.

21

Historical Background

2003:



The committee publishes the Haskell Report, defining a stable version of the language; an updated version was published in 2010.

22

Historical Background

2010-date:

Standard distribution, library support, new language features, development tools, use in industry, influence on other languages, etc.

23

Che algoritmo è?

```
void f(int a[], int lo, int hi)
{ int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);
    a[hi] = a[l];
    a[l] = p;
    f( a, lo, l-1 );
    f( a, l+1, hi );
  }
}
```

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a ← xs, a ≤ x]
              zs = [b | b ← xs, b > x]
```

24

Niente effetti collaterali

`int count( List l );`

- Accetta una lista e restituisce un intero
- Potrebbe anche modificare la lista
- Potrebbe modificare qualche variabile globale
- Potrebbe cancellare dei file dal disco rigido
- Potrebbe accettare connessioni da un hacker
- Potrebbe installare un virus…

`count :: List -> Int`

- Accetta una lista e restituisce un intero

25

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell, and comprises a compiler and interpreter;
- GHC is freely available from:

`www.haskell.org/platform`

26

Starting GHCi

The interpreter can be started from the terminal command prompt by simply typing ghci:

```
$ ghci

GHCi, version X: http://www.haskell.org/ghc/  :? for help

Prelude>
```

The GHCi prompt > means that the interpreter is now ready to evaluate an expression.

27

For example, it can be used as a desktop calculator to evaluate simple numeric expressions:

```
> 2+3*4
14
> (2+3)*4
20
> sqrt (3^2 + 4^2)
5.0
```

```
> 2^1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

The Standard Prelude

Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

- Select the first element of a list:

```
> head [1,2,3,4,5]
1
```

29

- Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Select the nth element of a list:

```
> [1,2,3,4,5] !! 2
3
```

- Select the first *n* elements of a list:

```
> take 5 [1,2,3,4,5,6,7,8]
[1,2,3,4,5]
```

30

Remove the first *n* elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

Calculate the length of a list:

```
> length [1,2,3,4,5]
5
```

Calculate the sum of a list of numbers:

```
> sum [1,2,3,4,5]
15
```

31

Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

32

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

$f(a,b) + c\,d$

Apply the function f to a and b, and add the result to the product of c and d.

33

In Haskell, function application is denoted using space, and multiplication is denoted using *.

```
f a b + c*d
```

As previously, but in Haskell syntax.

34

Moreover, function application is assumed to have higher priority than all other operators.

```
f a + b
```

Means (f a) + b, rather than f (a + b).

Similar to mathematical notation:

$\cos\ a + b$

means

$\cos(a) + b$

Otherwise, we write

$\cos(a + b)$

35

Examples

Mathematics	Haskell
f(x)	`f x`
f(x,y)	`f x y`
f(g(x))	`f (g x)`
f(x,g(y))	`f x (g y)`
f(x)g(y)	`f x * g y`

36

Haskell Scripts

- As well as the functions in the standard library, you can also define your own functions;
- New functions are defined within a script, a text file comprising a sequence of definitions;
- By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes.

37

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi.

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x    = x + x

quadruple x = double (double x)
```

38

Leaving the editor open, in another window start up GHCi with the new script:

```
$ ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

39

- Haskell lets you use both infix and prefix notation

Prefix	Infix

`> mod 6 4`
`2`
`> div 6 4`
`1`
	`> 6 `mod` 4`
	`2`
	`> 6 `div` 4`
	`1`

`> (*) 3 5`
`15`
`> (-) 6 4`
`2`
	`> 3 * 5`
	`15`
	`> 6 - 4`
	`2`

40

Leaving GHCi open, return to the editor, add the following two definitions, and resave:

```
factorial n = product [1..n]

average ns = sum ns `div` length ns
```

Note:

- **div** is enclosed in back quotes, not forward;
- **x `f` y** is just syntactic sugar for **f x y**.

41

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```

42

Useful GHCi Commands

Command	Meaning
:load *name*	load script *name*
:reload	reload current script
:set editor *name*	set editor to *name*
:edit *name*	edit script *name*
:edit	edit current script
:type *expr*	show type of *expr*
:?	show all commands
:quit	quit GHCi

43

Naming Requirements

- Function and argument names must begin with a lower-case letter. For example:

```
myFun    fun1    arg_2    x'
```

- By convention, list arguments usually have an s suffix on their name. For example:

```
xs    ns    nss
```

44

(1) Show how the library function last that selects the last element of a list can be defined using the functions introduced in this lecture.

(2) Can you think of another possible definition?

(3) Similarly, show how the library function init that removes the last element from a list can be defined in two different ways.

Part of these slides were adapted from the material of the book

Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016