Programmazione funzionale

Leggi il capitolo 13 di Linguaggi di programmazione – Principi e paradigmi

Computazione senza Stato

- Tutti i linguaggi convenzionali basano il loro modello computazionale sulla trasformazione dello stato
- Il cuore di questo modello è il concetto di *variabile modificabile*, cioè un contenitore con un nome a cui, durante il calcolo, possiamo assegnare valori diversi, mantenendo sempre la stessa associazione nell'ambiente.
- Il costrutto principale nei linguaggi convenzionali è *l'assegnamento*, che modifica il valore contenuto in una variabile

Computazione senza Stato

- Questa è una visione astratta della macchina fisica convenzionale sottostante.
- Il calcolo procede modificando i valori memorizzati nelle posizioni.
- Si chiama *macchina di von Neumann*, dal nome del matematico ungherese-americano che, negli anni '40, vide che la macchina di Turing poteva essere ingegnerizzata in un prototipo fisico, originando così il computer moderno

Computazione senza Stato

- Questo non è, tuttavia, l'unico modello possibile su cui basare un linguaggio di programmazione.
- È possibile calcolare senza utilizzare variabili modificabili
- Il calcolo procede non modificando lo stato ma *riscrivendo le espressioni*, cioè mediante cambiamenti che avvengono solo nell'ambiente e non coinvolgono il concetto di memoria.
- L'intero calcolo sarà espresso in termini di sofisticata modifica dell'ambiente

Computazione senza Stato

- Senza assegnazione, l'iterazione perde anche il suo vero senso.
- Un ciclo può modificare ripetutamente lo stato solo finché i valori di determinate variabili soddisfano una guardia.
- I costrutti iterativi e ricorsivi sono due meccanismi che consentono calcoli infinitamente lunghi (e potenzialmente divergenti).
- Nel modello computazionale senza stato, l'iterazione scompare, la ricorsione rimane e diventa il costrutto fondamentale per il controllo della sequenza.
- Le funzioni di ordine superiore e la ricorsione sono gli ingredienti di base di questo modello computazionale senza stato.

Paradigma di programmazione funzionale

- Tra i linguaggi di programmazione che presuppongono questo modello vi sono i *linguaggi funzionali* e il paradigma che ne deriva è chiamato *paradigma di programmazione funzionale*.
- Dagli anni '30, accanto alla Macchina di Turing, esiste il λ-calcolo, un modello astratto per caratterizzare le funzioni calcolabili che si basa esattamente sul concetto che abbiamo brevemente spiegato.
- Questo modello è più vicino alla matematica: dal punto di vista matematico scrivere x=x+1, come posso fare in un linguaggio con l'assegnamento, non ha senso
- LISP è stato il primo linguaggio di programmazione esplicitamente ispirato al λ-calcolo e molti altri ne sono seguiti negli anni (Scheme, ML, in tutti i suoi diversi dialetti, Miranda, Haskell, solo per citare i più comuni).
- Tra questi, solo Miranda e Haskell sono "puramente funzionali"; gli altri hanno anche componenti imperativi

Paradigma di programmazione funzionale

- Haskell è stato scelto come esempio perché, per la sua coerenza ed eleganza di design, è il più adatto ad una presentazione didattica.

Espressioni e Funzioni

- Nella consueta pratica matematica, c'è qualche ambiguità su quando definiamo una funzione e quando la applichiamo a un valore. Non è raro imbattersi in espressioni del genere:
- Sia $f(x) = x^2$ la funzione che associa a x il suo quadrato. Se ora abbiamo $x = 2$, segue che $f(x) = 4$.
- L'espressione sintattica $f(x)$ è usata per denotare due cose ben diverse: la prima volta serve per introdurre il *nome*, *f*, di una specifica funzione; la seconda volta serve a denotare il risultato *dell'applicazione* della funzione *f* ad un valore specificato.

Espressioni e Funzioni

- È opportuno distinguere i due casi
- Quando un matematico afferma di definire la funzione, *f (x)*, in realtà sta definendo la funzione *f* con un parametro formale, *x*, che serve a indicare la trasformazione che *f* applica al suo argomento.
- Per distinguere linguisticamente il nome e il "corpo" della funzione, seguendo la sintassi Haskell, possiamo scrivere:

```
f x = x*x
```

Espressioni e Funzioni

- Il nome *f* è legato alla trasformazione di *x* in *x* * *x*.
- In tutti i linguaggi funzionali, le funzioni sono valori *esprimibili*; possono cioè essere il risultato della valutazione di un'espressione complessa.
- Nel nostro caso, l'espressione a destra di = è un'espressione che denota una funzione.

Espressioni e Funzioni

- Per l'applicazione di una funzione ad un argomento, manteniamo la notazione tradizionale, scrivendo f(2) o (f 2) , oppure f 2 , per l'espressione che risulta dall'applicazione di una funzione f all'argomento 2.
- È possibile anche definire nuovi nomi nello stesso modo con cui si definiscono le funzioni, ad esempio:

```
square = f
four = f 2
```

Haskell in laboratorio

```
$ ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> f x = x*x
Prelude> f 2
4
```

- Per installare ghci sul vostro pc dovete installare GHCup che trovate a https://www.haskell.org/ghcup/
- Se avete linux, potete installare GHC dal package manager usando le istruzioni a https://www.haskell.org/downloads/

Espressioni e Funzioni

- L'introduzione di una sintassi specifica per un'espressione che denota una funzione ha una conseguenza importante.
- È possibile scrivere (ed eventualmente applicare) una funzione senza doverle necessariamente assegnare un nome. Ad esempio, l'espressione

```
(\x -> x + 1) (6)
```

ha il valore 7 che risulta dall'applicazione della funzione (anonima) (\x -> x + 1) all'argomento 6.

Espressioni e Funzioni

- Supponiamo (come fa Haskell) che l'applicazione possa essere denotata da una semplice giustapposizione (cioè senza parentesi) e che associa a sinistra (notazione prefissa). Se g è il nome di una funzione,
  g a1 a2 ... ak
  significa:
  (... ((g a1) a2)... ak).
- Nulla impedisce a un'espressione funzionale di apparire all'interno di un'altra, come in
  add x = (\y -> x + y)
- Il valore add è una funzione che, dato un argomento, x, restituisce una funzione anonima che, dato un argomento y, restituisce x+y

Espressioni e Funzioni

- Possiamo usare add in molti modi diversi:
  add 1 2
  3
  addtwo = add 2
  addtwo 3
  5
  Si noti che, in particolare, addtwo è una funzione che si ottiene come risultato della valutazione di un'altra espressione.

Espressioni e Funzioni

- Un esempio di funzioni che manipolano le funzioni, questa volta sotto forma di parametro formale, si consideri la definizione:
  comp f g x = f(g(x))
- Restituisce la funzione composta dai suoi primi due argomenti (che sono, a loro volta, funzioni).
- Si può anche usare la notazione seguente:
  comp f g x = (f . g) x

Espressioni e Funzioni

- Ogni programma funzionale permette la definizione di funzioni ricorsive.
- Usando *un'espressione condizionale* (che in Haskell è scritta con la sintassi familiare if then else), possiamo definire il solito fattoriale come: fatt n = if n==0 then 1 else n*fatt(n-1)

Calcolo come Riduzione

- Se escludiamo le funzioni aritmetiche (che possiamo assumere predefinite con la semantica usuale) e l'espressione condizionale, a livello concettuale, possiamo descrivere la procedura utilizzata per trasformare un'espressione complessa nel suo valore (*valutazione*) come processo di *riscrittura*
- Chiamiamo questo processo *riduzione*.
- In un'espressione complessa, una sottoespressione della forma "funzione applicata a un argomento" viene sostituita testualmente dal corpo della funzione in cui il parametro formale è sostituito, a sua volta, dal parametro effettivo.

Calcolo come Riduzione

- fact 3 → (fact n = if n==0 then 1 else n*fact(n-1)) 3
  → if 3==0 then 1 else 3*fact(3-1)
  → 3*fact(3-1)
  → 3*fact(2)
  → 3*((fact n = if n==0 then 1 else n*fact(n-1)) 2)
  → 3*(if 2==0 then 1 else 2*fact(2-1))
  → 3*(2*fact(2-1))
  → 3*(2*fact(1))
  → 3*(2*((fact n = if n==0 then 1 else n*fact(n-1)) 1))
  → 3*(2*(if 1==0 then 1 else 1*fact(1-1))
  → 3*(2*(1*fact(1-1))
  → 3*(2*(1*fact(0))
  → 3*(2*(1*((fact n = if n==0 then 1 else n*fact(n-1)) 0)))
  → 3*(2*(1*(if 0==0 then 1 else n*fact(n-1))))
  → 3*(2*(1*1))
  → 6

Calcolo come Riduzione

- Data la definizione
  r x = r(r(x));
  ogni calcolo che comporta una valutazione di *r* si risolve in una riscrittura infinita.
- Diciamo, in tal caso, che il calcolo *diverge* e che il risultato è indefinito.

Gli Ingredienti Fondamentali

- Dal punto di vista sintattico, un linguaggio come quello in esame non ha comandi (non essendoci stato da modificare usando effetti collaterali) ma solo espressioni.
- Oltre ai possibili valori e agli operatori primitivi per i dati (come integer, boolean, caratteri, ecc.) e l'espressione condizionale, i due costrutti principali per la definizione delle espressioni sono:
- *Abstraction* che, data qualsiasi espressione, exp e un identificatore, x, permette la costruzione di un'espressione \x -> exp
- *L'applicazione* di un'espressione, f_exp, ad un'altra espressione, a_exp, che scriviamo f_exp a_exp,

Gli Ingredienti Fondamentali

- Non ci sono vincoli sulle possibilità di passare funzioni come argomenti ad altre funzioni, o di restituire funzioni come risultati di altre funzioni (di ordine superiore).
- C'è una perfetta *omogeneità* tra programmi e dati.
- Dal punto di vista semantico, un programma è costituito da una serie di definizioni di valori, ognuna delle quali inserisce una nuova associazione nell'ambiente e può richiedere la valutazione di espressioni arbitrariamente complesse.
- La presenza di funzioni di ordine superiore e la possibilità di definire funzioni ricorsive rende questo meccanismo di definizione flessibile e potente.

Gli Ingredienti Fondamentali

- Ad una prima approssimazione, la semantica del calcolo (*valutazione*) non si riferisce ad aspetti linguistici diversi da quelli introdotti finora.
- Può essere definito utilizzando una semplice riscrittura simbolica delle stringhe (*riduzione*), che utilizza ripetutamente due operazioni principali per semplificare le espressioni fino a raggiungere una forma semplice che denota immediatamente un valore.
- La prima di queste operazioni è la semplice ricerca attraverso l'ambiente.
- La seconda operazione, che è più interessante, riguarda un'espressione funzionale applicata a un argomento (che è chiamata β-rule).

Definizione

- Un *redex* (che sta per *un'espressione riducibile*) è un'applicazione della forma ((\x -> body) arg) .
- Il *reductum* di un *redex* ((\x -> body) arg) è l'espressione che si ottiene sostituendo nel corpo ogni occorrenza (libera) del parametro formale, x, con una copia di arg (evitando l'acquisizione variabile).
- *β*-rule: Un'espressione, exp, in cui un redex appare come una sottoespressione, viene ridotta (o riscritta, semplifica) a exp1 (notation: exp $\rightarrow$ exp1), dove exp1 è ottenuto da exp sostituendo il redex con il suo reductum.
- Nel contesto della programmazione funzionale, gli identificatori associati ai valori sono spesso indicati come "variabili"

Valutazione

- Qual è la condizione risolutiva per la riduzione (cioè cosa significa “una forma semplice che denota immediatamente un valore”?)
- Quale semantica precisa deve essere data alla β-rule: quale ordine seguire durante la riscrittura dovrebbe essere presente più di un redex nella stessa espressione?

Valori

- Un *valore* è un'espressione che non può essere ulteriormente riscritta.
- In un linguaggio funzionale, ci sono valori di due tipi: valori di tipo primitivo e funzioni.
- Negli esempi precedenti abbiamo terminato la valutazione quando abbiamo raggiunto valori primitivi di tipo intero: 3 , 2 , 1.

Valori funzionali

- Consideriamo la seguente definizione:
  g x = ((\ y -> y+1) 2);
- Una definizione comporta la valutazione dell'espressione a destra dell'uguaglianza e il legame del valore così derivato al nome a sinistra del =.
- Ma, in questo caso, non è immediatamente chiaro quale sia il valore da associare a g. Abbiamo:
- \x -> 3
  in cui abbiamo riscritto il corpo di g valutando il redex che contiene,
  oppure abbiamo
  \x -> ((\y -> y+1) 2)
- ?

Valori funzionali

- Il primo caso sembra essere quello che più rispetta la semantica informale che abbiamo dato
- Il secondo, d'altra parte, è più vicino alla solita definizione di funzione in un linguaggio convenzionale, in cui il corpo di una funzione viene valutato solo quando viene chiamato.
- È il secondo che viene adottato per tutti i linguaggi funzionali di uso comune.
- La valutazione non avviene "sotto" un'astrazione.
- Ogni espressione della forma
  `\x -> exp`
  rappresenta un valore, quindi i redex eventualmente contenuti in exp *non vengono mai* riscritti finché l'espressione non viene applicata a qualche argomento

Sostituzione senza acquisizione

- La sostituzione deve essere capture-free
- Per la descrizione elementare che stiamo dando, è sufficiente una convenzione sintattica.
- In ogni espressione, non ci sono mai due parametri formali con lo stesso nome e i nomi delle possibili variabili che non sono parametri formali sono tutti distinti da quelli dei parametri formali.

Strategie di valutazione

- Ogni linguaggio deve fissare una strategia specifica (cioè un ordine fisso) per la valutazione delle espressioni.
- La presenza di funzioni di ordine superiore nei linguaggi funzionali rende questo requisito ancora più fondamentale
- k x y = x
  r z = r(r(z))
  d u = if u==0 then 1 else u
  succ v = v+1
  v = k (d (succ 0)) (r 2)
- Quale valore è associato a v e come viene determinato?

Strategie di valutazione

- La β-rule da sola non è di grande utilità perché, nella parte destra della definizione di v, ci sono 4 redex:
  k (d (succ 0))
  d (succ 0)
  succ 0
  r 2

Strategie di valutazione

- Ogni linguaggio comunemente usato usa una strategia che riduce i redex a partire da quello all'estremità più a sinistra.
- Qual è il redex più a sinistra di
  k (d (succ 0))
  d (succ 0)
  succ 0
- ?
- Questi tre redex sono sovrapposti l'uno all'altro.

Strategie di valutazione

- Dopo aver fissato un ordine di valutazione più a sinistra, scopriamo quindi che abbiamo tre diverse strategie.
- **Valutazione per valore**. Nella valutazione per valore (che è anche chiamata valutazione *nell'ordine applicativo* o valutazione *eager*, o valutazione *innermost*), un redex viene valutato solo se l'espressione che costituisce il suo argomento è già un valore.
- **Valutazione per nome**. Nella strategia di valutazione per nome (che è anche chiamata *ordine normale* o *outermost*), la parte funzione di un redex viene valutata prima della sua parte di argomento.
- **Valutazione pigra (lazy)**

Valutazione per valore

- La valutazione più a sinistra in ordine applicativo funziona come segue.
- Scansiona l'espressione da valutare da sinistra, scegliendo la prima applicazione incontrata. Sia (f_exp a_exp) .

1. Per prima cosa valuta (applicando ricorsivamente questo metodo) f_exp fino a quando non è stato ridotto a un valore (di tipo funzionale) della forma (\x -> ...) .
2. Quindi valuta la parte dell'argomento, a_exp, dell'applicazione, in modo che sia ridotta a un valore, val.
3. Infine, riduci il redex ((\x -> ...)val) usando la *β*-rule e ricomincia da (1)

Esempio

k (d (succ 0))
Prima sceglie l'applicazione k (d (succ 0))

- Alcune applicazioni elementari di (1), (2) e (3) serviranno a dimostrare che k, d e succ sono già valori
- Il primo redex da ridurre è quindi succ 0 (Cioè, ((\ v -> v + 1) 0)) che sarà completamente valutato
  e produce 1.
- Quindi il redex (d 1) (Cioè ((\u -> if u==0 then 1
  else u) 1)) è ridotto per dare il valore 1.

```
k x y = x
r z = r(r(z))
d u = if u==0 then 1
else u
succ v = v+1
v = k (d (succ 0)) (r 2)
```

Esempio

- Quindi (k 1) viene valutato per produrre il valore (\y -> 1)
- L'espressione è diventata
  (\y -> 1) (r 2)
- Poiché la parte funzionale di questa applicazione è già un valore, la strategia prescrive che l'argomento (r 2) venga valutato.
- La valutazione porta a riscrivere in r (r 2) , poi in r (r (r 2)) e così via in un calcolo divergente.
- Nessun valore è quindi associato a v perché il calcolo diverge.

```
k x y = x
r z = r(r(z))
d u = if u==0 then 1
else u
succ v = v+1
v = k (d (succ 0)) (r 2)
```

Valutazione per nome

- La valutazione più a sinistra in ordine normale procede come segue.

1. Scansiona l'espressione da valutare da sinistra, scegliendo la prima applicazione incontrata. Così sia (f_exp a_exp)
2. Per prima cosa valuta f_exp (applicando ricorsivamente questo metodo) fino a quando non è stato ridotto a un valore (di tipo funzionale) della forma (\x -> ...)
3. Riduci il redex ((\x -> ..) a_exp) utilizzando la *β*-rule e vai a (1).

Esempio

k (d (succ 0))
d (succ 0)
succ 0
r 2

```
k x y = x
r z = r(r(z))
d u = if u==0 then 1
else u
succ v = v+1
v = k (d (succ 0)) (r 2)
```

- Il primo redex da ridurre è quindi:
  k (d (succ 0))
- Viene riscritto in:
  \y -> d (succ 0)
  che è un valore funzionale.
- L'espressione ora è della forma:
  (\y -> d (succ 0)) (r 2)
  per il quale la strategia prescrive di ridurre il redex più esterno, quindi otteniamo:
  d (succ 0)

Esempio

- Ora riducendo questa espressione, otteniamo:
  if (succ 0)==0 then 1 else (succ 0)
- E poi:
  if 1==0 then 1 else (succ 0)
- e infine
  succ 0
- da cui otteniamo il valore finale, 1, che viene poi associato a v

```
k x y = x
r z = r(r(z))
d u = if u==0 then 1
else u
succ v = v+1
v = k (d (succ 0)) (r 2)
```

Valutazione Lazy

- Nella valutazione per nome, un singolo redex potrebbe dover essere valutato più di una volta a causa di alcuni doppioni che si sono verificati durante la riscrittura
- Nell'esempio il redex (succ 0) è duplicato a causa della funzione d e viene ridotto due volte nell'espressione condizionale che forma il corpo di d.
- Questo è il prezzo che deve essere pagato per posticipare la valutazione di un argomento a dopo l'applicazione di una funzione
- Ma è molto costoso in termini di efficienza (quando il redex duplicato richiede una quantità significativa di calcolo).

Valutazione Lazy

- Per ovviare a questo problema e mantenere i vantaggi della valutazione per nome, la strategia lazy procede così per nome, ma la prima volta che si incontra una "copia" di un redex, il suo valore viene salvato e verrà utilizzato nel caso in cui si incontrino altre copie dello stesso redex.
- Le strategie *per nome* e *lazy* sono esempi delle strategie *chiama per necessità* (*call-by-need*) in cui un redex viene ridotto solo se richiesto dal calcolo.

Confronto delle strategie

- È sensato chiedersi se le strategie per valore e per nome producano valori *distinti* per la stessa espressione.
- Diciamo che un'espressione nel linguaggio è *chiusa* se tutte le sue variabili sono vincolate da qualche \.

Teorema

- *Sia* exp *un'espressione chiusa. Se* exp *si riduce a un valore primitivo,* val*, usando una delle tre strategie, allora* exp *si riduce a* val *seguendo la strategia per nome. Se* exp *diverge usando la strategia per nome, allora diverge anche sotto le altre due strategie.*
- Una delle caratteristiche più importanti del paradigma funzionale *puro*.
- Il teorema esclude il caso in cui un'espressione (chiusa) possa produrre un valore primitivo, val1, in una strategia e produrre *un altro* valore primitivo, val2 , usando un'altra strategia
- Due strategie possono quindi differire solo per il fatto che una determina un valore mentre l'altra diverge

Teorema

- La seguente proprietà di base è fondamentale per la validità del teorema:
- Una volta che una strategia è stata fissata in un determinato ambiente, la valutazione di tutte le occorrenze di una singola espressione produce sempre lo stesso valore
- Questa proprietà, che viene immediatamente falsificata quando ci sono effetti collaterali, è presa da molti autori come *criterio* per un linguaggio funzionale puro
- Questa è una proprietà molto importante che consente di *ragionare* su un programma funzionale

Commenti

- È proprio in virtù di questa proprietà che la valutazione lazy è corretta.
- Alla luce del teorema precedente, ci si può chiedere quale interesse ci sia nella strategia per-valore, dato che per-nome è la più generale di tutte le possibili strategie…
- Inoltre, le ragioni di efficienza sembrerebbero suggerire che adottiamo solo strategie di call-by-need, dato che non è chiaro perché i redex inutili dovrebbero essere ridotti
- Il punto è che il caso dell'efficienza non è così semplice.
- Implementare una strategia call-by-need è, in generale, più costoso di una semplice strategia per valore (call-by-value).

Commenti

- La strategia per valore ha implementazioni efficienti su architetture convenzionali, anche se, a volte, svolge un lavoro inutile (ogni volta che viene valutato un argomento che non è richiesto nel corpo della funzione)
- Quest'ultimo caso, tuttavia, può essere trattato in modo efficiente con strategie di valutazione astratte che tentano di identificare argomenti inutili.
- Tra i linguaggi funzionali LISP, Scheme e ML utilizzano una strategia per valore (anche perché includono importanti aspetti imperativi), mentre Miranda e Haskell (che sono linguaggi funzionali puri) usano la valutazione lazy.

Programmazione in linguaggio funzionale

- I meccanismi che abbiamo descritto nell'ultima sezione sono sufficienti per esprimere i programmi per ogni funzione calcolabile
- Questo nucleo di ogni linguaggio funzionale però è troppo austero per poterlo utilizzare come un vero e proprio linguaggio di programmazione.
- Ogni linguaggio funzionale, quindi, inserisce questo nucleo in un contesto più ampio che prevede meccanismi di diverso tipo, ognuno con l'obiettivo di rendere la programmazione più semplice ed espressiva.

Ambiente locale

- È opportuno prevedere meccanismi espliciti per introdurre definizioni di portata limitata, come ad esempio:
  let x = exp in exp1
- Questo introduce l'associazione di x al valore di exp in un ambito che include solo exp1.
- A dire il vero, la presenza di espressioni funzionali introduce già ambiti nidificati e ambienti associati che sono composti dai parametri formali della funzione legati ai parametri effettivi.
- Dal punto di vista della valutazione, possiamo infatti considerare il costrutto come zucchero sintattico per
  (\x -> exp1) exp.

Interattività

- Ogni linguaggio funzionale ha un ambiente interattivo.
- Il linguaggio viene utilizzato inserendo espressioni che la macchina astratta valuta e di cui restituisce il valore calcolato.
- Le definizioni sono espressioni particolari che modificano l'ambiente globale (e possono restituire un valore).
- Questo modello suggerisce immediatamente implementazioni basate su interprete, anche se esistono implementazioni efficienti che utilizzano la compilazione per generare codice compilato la prima volta che una definizione viene archiviata nell'ambiente.

Interattività

- GHCi fornisce un suo interprete, Prelude, che contiene un ampio numero di funzioni di libreria.
- Oltre alle funzioni numeriche familiari come + e *, la libreria include un ampio numero di funzioni sulle liste.

Scripts

- Oltre alle funzioni nella libreria standard di Prelude, si possono definire proprie funzioni in uno script, un file di testo contenente un insieme di definizioni
- Per convenzione, gli script Haskel hanno l'estensione .hs nel nome del file
- In un editor di testo, scrivere le seguenti due definizioni di funzione e salvarle nel file test.hs:

```
double x = x +x

quadruple x = double (double x)
```

- Lanciare GHCi con il nuvo script

```
$ ghci test.hs
```

- A questo punto le funzioni di test.hs saranno disponibili

```
Prelude> quadruple 10
40
```

Scripts

- Aggiungere una definizione a test.hs:
  triple x = x+x+x
- GHCi non si accorge che lo script è cambiato, occorre ricaricarlo esplicitamente con :reload
  Prelude> :reload
  Prelude> triple (double 3)
  18
- Inoltre, uno script può essere caricato con :load
  Prelude> :load test.hs
- Si può omettere l'estensione
  Prelude> :load test

Tipi

- Ogni linguaggio funzionale che abbiamo citato fornisce il tipi primitivi usuali (interi, booleani, caratteri) con operazioni sui loro valori.
- Ad eccezione di Scheme, che è un linguaggio con controllo dinamico dei tipi, tutti gli altri hanno elaborati sistemi di tipi statici.
- Questi sistemi di tipi consentono la definizione di nuovi tipi come coppie, elenchi, "record" (cioè tuple di valori etichettati).

Tipi

- Un tipo è un nome per una collezione di valori collegati
- Ad esempio, Haskell ha il tipo primitivo Book che contiene i due valori logici False e True
- Se, valutando un'espressione e si produce un valore di tipo t, allora e ha tipo t e si scrive
  e :: t
- Haskell ha i tipi primitivi

Bool	- valori logici
Char	- caratteri singoli
String	- stringhe di caratteri
Int	- interi a singola precision
Integer	- interi a precisione arbitraria
Float	- numeri floating point

Operatori

- Vediamo alcuni operatori per i vari tipi
- Bool:
  - not : not
  - && : and
  - || : or
- Operatori di confronto:
  - < : minore
  - > : maggiore
  - == : uguale
  - /= : diverso
  - <= : minore o uguale
  - >= : maggiore o uguale

Tipi

- In GHCi, il comando :type calcola il tipo di un'espressione, senza valutarla

```
Prelude> :type True
True :: Bool
Prelude> not False
True
Prelude> :type not False
not False :: Bool
```

- Si può anche usare esplicitamente un tipo

```
Prelude> 3 :: Int
3
Prelude> 3 :: Double
3.0
```

Liste

- Una lista è una sequenza di valori dello stesso tipo:
  [False, True, False] :: [Bool]
  [‘a’,’b’,’c’]::[Bool]

  Prelude> [1,2,3,”a”,”bb”,”ccc”]
  ERROR!
- In generale, [t] è una lista di elementi di tipo t
- È la struttura dati più comune in Haskell
- Gli elementi sono separati da virgole e racchiusi tra parentesi quadre
  Prelude> [3,1,5,3]
  [3,1,5,3]
  Prelude> [“list”,”of”,”strings”]
  [“list”,”of”,”strings”]

Liste

- Il tipo di una lista non dice niente sulla sua lunghezza:
  [False,True] :: [Bool]
  [False,True,False] :: [Bool]
- Non ci sono limiti sul tipo degli elementi, ad esempio si possono avere anche liste innestate:
  [[‘a’],[‘b’,’c’]] :: [[Char]]

Enumerazioni

- Inizia da 1 e termina a 10
  Prelude> [1..10]
  [1,2,3,4,5,6,7,8,9,10]
- Inizia da 1, conta incrementando di 0.25 fino a 3
  Prelude> [1,1.25..3.0]
  [1.0,1.25,1.5,1.75,2.0,2.25,2.5,2.75,3.0]
- Conto alla rovescia
  Prelude> [10,9..0]
  [10,9,8,7,6,5,4,3,2,1,0]

Operazioni sulle liste

- Concatenazione ++
  Prelude>[1,2,3]++[4,5,6]
  [1,2,3,4,5,6]
  Prelude>(++) [1,2,3] [4,5,6]
  [1,2,3,4,5,6]
- Costruisci :
  Prelude> 0 : [1,2,3]
  [0,1,2,3]
  Prelude> (:) 0 [1,2,3]
  [0,1,2,3]
- Molto efficiente
- Infatti una lista è formalmente definita per mezzo di
  Prelude> [1,2,3] == (:) 1 ((:) 2 ((:) 3 []))
  True
- [] lista vuota

Operazioni sulle liste

- head
  Prelude>head [1,2,3]
  1
- tail
  Prelude>tail [1,2,3]
  [2,3]
- take
  Prelude>take 2 [1,2,3]
  [1,2]
- drop
  Prelude> drop 2 [1,2,3]
  [3]

Stringhe

- Una stringa non è altro che una lista di caratteri
  Prelude> “wahoo” == [‘w’,’a’,’h’,’o’,’o’]
- Quindi (++) e (:) funzionano anche sulle stringhe

Tuple

- Una tuple è una sequenza di valori di tipo diverso
  (False,True) :: (Bool,Bool)
  (False,'a',True) :: (Bool,Char,Bool)
- In generale (t1,t2,…,tn) è il tipo delle n-tuple il cui i-esimo componente ha tip ti for ogni i da 1 a n
- Il tipo di una tuple codifica la sua lunghezza
  (False,True) :: (Bool,Bool)
  (False,True,True) :: (Bool,Bool,Bool)
- Ciascun componente può avere qualunque tipo
  ('a',(False,'b')) :: (Char,(Bool,Char))
  (True,['a','b']) :: (Bool,[Char])
- fst and snd: Prelude> fst (1,2) 1 Prelude> snd (1,2) 2
- Funzionano solo per tuple di due elementi

Tipi

- Una parte importante del sistema dei tipi nei linguaggi funzionali è quella dedicata ai tipi di funzioni
- Una funzione è un mapping da valori di un tipo a valori di un altro
  not :: Bool -> Bool
  even :: Int -> Bool
- In generale t1 -> t2 è il tipo delle funzioni che mappano valori di tipo t1 in valori di tipo t2
  Prelude> head [1,2,3,4]
  1
  Prelude> :type head
  head :: [a] -> a
  Prelude> :type fst
  fst :: (a,b) -> a
- a e b sono «variabili tipo», placeholders per qualunque tipo

Tipi

- Gli argomenti e i risultati possono essere di tipo qualunque
- Ad esempio, si possono definire funzioni usando le tuple o le liste

```
add :: (Int,Int) -> Int
add (x,y) = x+y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried functions

- Funzioni con argomenti multipli sono possibili ritornando funzioni come risultato
  add' :: Int -> (Int -> Int)
  add' x y = x+y
- add' prende un intero x e restituisce una funzione add' x.
- A sua volta add' x prende un intero y e restituisce x+y.
- add e add' producono lo stesso risultato ma add prende due argomenti allo stesso tempo mentre add' ne prende uno alla volta
  add :: (Int,Int)->Int
- Le funzioni che prendono gli argomenti uno alla volta sono chiamate «curried», in onore di H.B Curry che ha lavorato con queste funzioni

Curried functions

- Le funzioni con più di due argomenti possono essere curried restituendo funzioni innestate
  mult :: Int -> (Int -> (Int -> Int)
  mult x y z = x*y*z
- mult prende un intero x e restiuisce la funzione mult x che prende un intero y e restituisce la funzione mult x y, che infine prende un intero z e restituisce x*y*z
- Le funzioni curried sono più flessibili delle funzioni sulle tuple perché funzioni utili possono essere ottenute applicando parzialmente una funzione curried, per esempio:
  add' 1 :: Int -> Int
  take :: Int -> ([Int] -> Int)
  take 5 :: [Int] -> Int
  drop :: Int -> ([Int] -> Int)
  drop 5 :: [Int] -> Int

Curried functions

- Per evitare un eccesso di parentesi si assume che la freccia -> sia associativa a destra
  Int -> Int -> Int -> Int
- significa
  Int -> (Int -> (Int -> Int))
- Di conseguenza, l'applicazione di funzione associa a sinistra
  mult x y z
- significa
  ((mult x) y) z
- Se non vengono esplicitamente usate tuple, tutte le funzioni in Haskell sono definite normalmente in forma curried

Funzioni polimorfiche

- Una funzione è detta polimorfica se il suo tipo contiene delle variabili di tipo
  length :: [a] -> Int
- Per qualunque tipo a, length prende una lista di a e restituisce un intero
  Prelude> length [True,False]
  2
- In questo caso a vale Bool
  Prelude> length [1,2,3,4]
  4
- In questo caso a vale Int
- Le variabili di tipo devono iniziare con una lettera minuscola e di solito sono chiamate a, b, c, …

Funzioni polimorfiche

- Molte funzioni definite nel prelude standard sono polimorfiche

```
fst     :: (a,b) -> a
head    :: [a] -> a
take    :: Int -> [a] -> [a]
zip     :: [a] -> [b] -> [(a,b)]
id      :: a -> a
```

Funzioni overloaded

- Una funzione polimorfica è detta overloaded se il suo tipo contiene uno o più vincoli di classe
  (+) :: Num a => a -> a -> a
- Per qualunque tipo *numerico* a, (+) prende due valori di tipo a e restituisce un valore di tipo a
- Le variabili di tipo vincolare possono essere istanziate con qualunque tipo che soddisfa il vincolo

```
Prelude> 1+2
3                          a=Int
Prelude> 1.0 + 2.0
3.0                        a=Float
Prelude> 'a'+'b'
ERROR!                     Char is not a numeric type
```

Funzioni overloaded

- I vincoli di classe includono

  Num tipi numerici

  Eq tipi di uguaglianza (tipi su cui è definita una relazione di uguaglianza)

  Ord tipi ordinati (tipi su cui è definita una relzione d'ordine)

- Per esempio

```
(+)     :: Num a => a -> a -> a
(==)    :: Eq a => a -> a -> Bool
(<)     :: Ord a => a -> a -> Bool
```

Suggerimenti

- Quando si definisce una nuova funzione in Haskell, è utile iniziare scrivendo il suo tipo
- In uno script, è buona pratica indicare il tipo di ogni funzione definita
- Quando si definisce il tipo di funzioni polimorfiche che usano numeri, uguaglianze e ordinamento, indicare il relativo vincolo di classe

Tipi

- Nei linguaggi funzionali tipizzati alcune espressioni funzionali sono illegali perché
  non possono essere tipizzate.
- Ad esempio, la seguente funzione è illegale in un linguaggio tipizzato:
  `f n = if n==0 then f(1) else f("pippo");`
- La ragione di ciò è che il parametro formale, f, deve avere contemporaneamente tipi Int -> a e String -> a (dove a denota una variabile di tipo, cioè un tipo generico che non è ancora stato istanziato).

Tipi

- Un'altra espressione illegale è l'autoapplicazione:
  delta x = x x
- L'espressione, (x x) , è illegale perché non c'è modo di assegnare un tipo univoco e coerente a x.
- Dato che si presenta a sinistra come applicazione, deve avere un tipo di forma a -> b.
- Poiché, quindi, appare anche come l'argomento di una funzione (a destra dell'applicazione), deve avere il tipo che la funzione richiede: pertanto, x deve essere del tipo a.
- Mettendo insieme i due vincoli, abbiamo che x deve, allo stesso tempo, essere di tipo a e di tipo a -> b e non c'è modo di "unificare" queste due espressioni.

Tipi

- Nei linguaggi senza un forte sistema di tipizzazione, come Scheme, la funzione delta è, invece, legale.
- In Scheme, possiamo anche applicare delta a se stesso.
- L'espressione (delta delta) costituisce un semplice esempio di programma divergente.
- La mancanza di un controllo statico del tipo rende possibile in Scheme scrivere espressioni come:
  (4 3)
- La macchina astratta genererà un errore a tempo di esecuzione.

```
delta x = x x
```

Pattern Matching

- Uno degli aspetti più fastidiosi della programmazione funzionale ricorsiva è la gestione dei casi terminali mediante espressioni "if" esplicite.
- Prendiamo, ad esempio, la funzione (inefficiente) che restituisce l'*n*° termine della serie di Fibonacci:

```
fibo n = if n==0 then 1
else if n==1 then 1
else fibo(n-1)+fibo(n-2)
```

Il meccanismo di *pattern matching* presente in alcuni linguaggi come ML e Haskell, ci permette di dare una definizione come segue:

```
fibo 0 = 1
fibo 1 = 1
fibo n = fibo(n-1)+fibo(n-2)
```

Pattern Matching

- Ogni ramo della definizione corrisponde a un caso diverso della funzione.
- La parte più interessante di questa definizione sono i parametri formali. Non sono più vincolati ad essere identificatori ma possono essere *pattern*, cioè espressioni formate da variabili, costanti e altri costrutti che dipendono dal sistema di tipi del linguaggio.
- Un pattern è una sorta di schema, un modello al quale far corrispondere il parametro effettivo.
- Quando la funzione viene applicata a un parametro effettivo, questo viene confrontato con i pattern e viene scelto il corpo corrispondente al primo pattern che corrisponde al parametro effettivo.

Pattern Matching

- In ghci per inserire definizioni su più righe bisogna racchiuderle in :{ e :}

```
Prelude> :{
Prelude| fibo 0=1
Prelude| fibo 1=1
Prelude| fibo n=fibo(n-1)+fibo(n-2)
Prelude| :}
Prelude> fibo 10
89
```

- Oppure, salvare il codice in un file fibonacci.hs e caricarlo con

```
Prelude> :load fibonacci.hs
```

Pattern Matching

- Il meccanismo di pattern matching è particolarmente flessibile quando si utilizzano tipi strutturati, ad esempio liste.
- Utilizzando il pattern matching, possiamo definire una funzione che calcola la lunghezza di un elenco generico come:
  length [] = 0
  length (e:rest)= 1 + length(rest)
- Si noti che i nomi utilizzati nei pattern vengono utilizzati come parametri formali per indicare parti del parametro effettivo.
- Dovrebbe essere chiaro il vantaggio in termini di concisione e chiarezza rispetto alla definizione usuale:
  length list = if list==[] then 0
  else 1 + length(tail list)
  Questa definizione, inoltre, richiede l'introduzione di una funzione di *selezione* per ottenere la lista che si ottiene rimuovendo l'elemento head.

Pattern Matching

- Un vincolo importante è che una variabile non può apparire due volte nello stesso modello.
- Si consideri, ad esempio, la seguente "definizione" di una funzione che, applicata a una lista, restituisce True se e solo se i primi due elementi della lista sono uguali:
  eq [] = False
  eq [e] = False
  eq (x:x:rest) = True
  eq (x:y:rest) = False
- Questa definizione è sintatticamente illegale perché il terzo pattern contiene la variabile x due volte.
- Il pattern matching *non* è unificazione

Pattern Matching

- Definizione dell'operatore and && (è predefinito ma vediamo lo stesso una possible definizione):
  True && True = True
  True && False = False
  False && True = False
  False && False = False
- Può essere definito in forma più compatta come
  True && True = True
  _ && _ = False
- _ indica una variabile senza nome (non si verifica il vincolo della diversità delle variabili)

Guarded equations

- In alternativa ai condizionali, le funzioni possono essere definite usando guarded equations
- Una guardia è introdotta dal simbolo |, è seguita da una espressione booleana e poi dal corpo della funzione

```
abs n | n >= 0          = n
      | othwerwise = -n
guessMyNumber x
  | x>27              = “Too large”
  | x<27              = “Too small”
  | otherwise         = “Correct”
signum n | n<0              = -1
         | n==0             = 0
         | otherwise    = 1
```

- otherwise equivale a True

Uso di let

- let introduce variabili immutabili
- Una volta definite non posso cambiare
  slope (x1,y1) (x2,y2) = let dy=y2-y1 dx=x2-x1 in dx/dy
- Si può usare anche la parola chiave where
  slope (x1,y1) (x2,y2) = dx/dy where dy=y2-y1 dx=x2-x1
- I binding where possono essere condivisi da guardi multiple

```
bmitTell weight height
  | bmi <= 18.5 = "underweight"
  | bmi <= 25.0 = "normal"
  | bmi <= 30.0 = "fat"
  | otherwise = "very fat"
where bmi = weight/height^2
```

Uso di let

- Le clausole let sono esse stesse espressioni
  ```
  Prelude> 4* (let a = 9 in a+1) +2
  42
  ```
- Possono anche essere usate per introdurre funzioni nello scope locale
  ```
  Prelude> [let square x = x*x in (square 5, square 3)]
  [(25,9)]
  ```

Oggetti infiniti

- In presenza di strategie per-nome o lazy, è possibile definire e manipolare *flussi*, cioè strutture di dati che sono (potenzialmente) infinite.
- Daremo un piccolo esempio di come questo può essere fatto.

Oggetti infiniti

- In primo luogo, dobbiamo chiarire il concetto di valore per le strutture di dati come le liste.
- In un linguaggio che utilizza la valutazione eager, un valore di lista di tipo T è una lista i cui elementi sono *valori* di tipo T.
- In un linguaggio lazy, questa non è una buona nozione di valore perché potrebbe richiedere la valutazione di redex inutili, contrariamente alla filosofia del call-by need.
- Per vedere il motivo, si definiscano le funzioni:
  hd (x:rest) = x
  tl (x:rest) = rest
- Le funzioni restituiscono, rispettivamente, il primo elemento (cioè la testa) e il resto (cioè la coda) di una lista non vuota

Oggetti infiniti

- Consideriamo ora l'espressione
  hd [2, ((\n -> n+1) 2)]
- Per calcolare il suo valore (cioè 2), non è necessario ridurre il redex che comprende il secondo elemento dell'elenco.
- Per questi motivi, in un contesto di call-by-need, un valore di tipo lista è qualsiasi espressione della forma: exp1 : exp2
  dove exp1 ed exp2 possono contenere anche redex.
- È quindi possibile definire una lista per ricorsione, come ad esempio:
  infinity2 = 2 : infinity2
- L'espressione infinity2 corrisponde a una lista potenzialmente infinita i cui elementi sono tutti 2

Oggetti infiniti

- Questo valore è perfettamente manipolabile sotto valutazione pigra. Per esempio,
  hd infinity2
- è un'espressione la cui valutazione termina con il valore 2 .
- Un'altra espressione con valore 2 è
  hd (tl (tl (tl infinity2)))

Oggetti infiniti

- Come ultimo esempio di flusso, la seguente funzione costruisce un elenco infinito di numeri naturali che inizia con il suo argomento, n:
  numbersFrom n = n : numbersFrom(n+1)
- Possiamo definire una funzione di ordine superiore che applica il suo argomento funzionale a tutti gli elementi di una lista, come in:
  map f [] = []
  map f (e:rest) =(f(e):map f rest)
- Ora possiamo produrre la lista infinita di tutti i quadrati partendo da n*n:
  squaresFrom n = map (\y -> y*y) (numbersFrom n)
- Otteniamo:
  Prelude> hd(squaresFrom 10)
  100

Il paradigma funzionale: una valutazione

- Dove risiede l'interesse per i linguaggi funzionali?
- Tale domanda deve essere posta su almeno due piani diversi:
  - Quella della programmazione pratica.
  - Quello della progettazione del linguaggio di programmazione.

Correttezza del programma

- Il principiante informatico spesso sostiene che la progettazione e la scrittura di un programma efficiente sono le operazioni più importanti su cui si basa la loro occupazione.
- L'ingegneria del software, tuttavia, ha ampiamente dimostrato, sia in teoria, come in ampi studi sperimentali, che i fattori più critici in un progetto software sono la sua correttezza, leggibilità, manutenibilità e la sua affidabilità.
- In termini economici, questi fattori rappresentano oltre il cinquanta per cento del costo totale; in termini sociali, la manutenzione del software può coinvolgere centinaia di persone diverse per un periodo di decine di anni. In termini etici, la vita, o la salute, di molte centinaia di persone può dipendere da un sistema software.

Correttezza del programma

- È ancora lontano il tempo in cui saremo in grado di produrre software con garanzie di correttezza paragonabili a quelle con cui un ingegnere civile rilascia i propri prodotti (ponti, colonne, strutture).
- L'ingegnere civile, infatti, ha a disposizione un intero corpus di matematica applicata con cui "calcolare" le strutture.
- Il progettista del sistema informativo è ben lontano dall'avere a sua disposizione una matematica anche lontanamente alla pari con quelle a disposizione del progettista edile.

Correttezza del programma

- Uno dei motivi del ritardo nel riuscire a produrre un tale livello di garanzia di correttezza è che ragionare su programmi con effetti collaterali è particolarmente difficile e costoso in termini di tempo di calcolo.
- Al contrario, esistono tecniche standard, basate su opportune varianti dell'induzione, che consentono di ragionare sui programmi privi di effetti collaterali.
- Ecco, dunque, un primo motivo per lo studio dei linguaggi funzionali puri.
- Se l'affidabilità, la leggibilità, la correttezza sono più importanti dell'efficienza, non c'è dubbio che la programmazione funzionale genera software più leggibile, la cui correttezza è più facile da stabilire e che, quindi, è più affidabile.

Schema del programma

- Le funzioni di ordine superiore, che sono comunemente usate nel paradigma funzionale, hanno importanti vantaggi pragmatici.
- Un modo tipico di utilizzare l'ordine superiore è quello di sfruttarlo per definire *schemi di programmazione generali* da cui è possibile ottenere programmi specifici mediante istanziazione.
- Consideriamo, ad esempio, un semplice programma che somma gli elementi di una lista di numeri interi. Possiamo scrivere

```
addl [] = 0
addl (n:rest )= n + addl(rest)
```

Se ora vogliamo il prodotto di una lista, possiamo scrivere:

```
prodl [] = 1
prodl (n:rest )= n * prodl(rest)
```

foldr

- È chiaro che questi due programmi sono istanze di un singolo schema di programma,
  che è generalmente chiamato *foldr*:
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f i [] = i
  foldr f i (n:rest) = f n (fold f i rest)
- Qui, f è l'operazione binaria da applicare agli elementi successivi della lista (+ o ∗), i è il valore da utilizzare nel caso terminale (l'elemento neutro dell'operazione) e il terzo parametro è l'elenco su cui eseguire l'iterazione:
- foldr ⊕ i [x1,…,xn] = x1⊕(x2⊕(…(xn⊕i)))
  Prelude> foldr (+) 0 [2,3,4]
  9
  Prelude> foldr (*) 1 [2,3,4]
  24

foldr

- Possiamo immediatamente vedere che possiamo definire le nostre due funzioni usando *foldr*:

```
addl = foldr (+) 0
prodl = foldr (*) 1
Prelude> addl [2,3,4]
9
Prelude> prodl [2,3,4]
24
```

- Qui, il fatto che la funzione sia di ordine superiore viene utilizzato sia per passare a foldr la funzione per iterare sull'elenco, sia per fare si che fold (+) 0 restituisca una funzione che richiede un argomento di tipo lista.
- L'uso estensivo degli schemi di programma aumenta la modularità del codice e consente di fattorizzare le prove di correttezza.

foldl

- C'è anche un left fold
  foldl :: (a -> b -> a) -> a -> [b] -> a
- La differenza rispetto a foldr è che il valore è accumulato a sinistra
  foldr ⊕ i [x1,…,xn] = x1⊕(x2⊕(…(xn⊕i)))
  foldl ⊕ i [x1,…,xn] = (((i⊕x1)⊕x2)…⊕xn)
- Definizione
  foldl f i [] = i
  foldl f i (n:rest) = foldl (f i n) rest

foldl

- Con il + e il * il risultato non cambia perché sono commutative, con il / invece:

```
foldl (/) 1 [1, 2, 3]
=> / (/ (/ 1 1) 2) 3
=> / (/ 1 2) 3
=> / 0.5 3
=> 0.16666667
```

```
foldr (/) 1 [1, 2, 3]
=> / 1 (/ 2 (/ 3 1))
=> / 1 (/ 2 3)
=> / 1 0.666666667
=> 1.5
```

filter

- filter è una funzione di ordine superiore che seleziona gli elementi di una lista che soddisfano un predicato
  filter :: (a -> Bool) -> [a] -> [a]
- Per esempio
  Prelude> filter even [1..10]
  [2,4,6,8,10]
- Può essere definita come

```
filter p [] = []
filter p (x:xs)
  | p x              = x : filter p xs
  | otherwise        = filter p xs
```

Valutazione

- Gli argomenti che abbiamo addotto a favore dei linguaggi funzionali non provengono solo da ambienti accademici.
- Probabilmente la difesa più appassionata del paradigma funzionale è quella della conferenza di accettazione del Turing Award di John Backus del 1977 (il premio è stato conferito per il suo lavoro su FORTRAN).
- Anticipando i tempi, in un certo senso, Backus ha messo i concetti di correttezza e leggibilità al centro del processo di produzione del software, relegando l'efficienza al secondo posto.
- Ha anche identificato la programmazione funzionale *pura* come lo strumento con cui procedere nella direzione che ha sostenuto.

Valutazione

- Oggi abbiamo macchine molto più efficienti di quelle del 1977, ma non abbiamo fatto i progressi necessari nel campo delle tecniche di correttezza del software.
- Nessuno ha tuttavia realmente sperimentato programmi puramente funzionali su larga scala in modo tale da poter fare un vero confronto con i programmi imperativi.
- Esperimenti significativi sono stati condotti presso IBM utilizzando il linguaggio FP definito da Backus e in centri di ricerca accademici utilizzando il linguaggio Haskell.

Valutazione

- Tutti gli altri linguaggi funzionali in uso presentano cospicui aspetti imperativi che, sebbene utilizzati in modi e luoghi molto diversi da quelli dei linguaggi imperativi ordinari, dissipano i vantaggi dei linguaggi funzionali in termini di dimostrazioni di correttezza
- In conclusione, dobbiamo dire che, dal punto di vista pratico della programmazione (intesa alla luce dell'intero processo di produzione del software), la superiorità di un paradigma sull'altro deve ancora essere dimostrata.

Progettazione del linguaggio di programmazione

- Gli studi e l'esperienza con i linguaggi funzionali hanno avuto un impatto significativo sulla progettazione dei linguaggi di programmazione.
- Molti singoli concetti ed esperimenti di programmazione funzionale sono successivamente migrati verso altri paradigmi.
- Tra questi concetti, il più noto al lettore è quello del sistema di tipi.
- I concetti di generici, polimorfismo, sicurezza dei tipi, hanno tutti avuto origine nei linguaggi funzionali (perché è più semplice studiarli e implementarli in un ambiente senza effetti collaterali).