Azzolini Riccardo 2021-03-04

Programmazione funzionale

1 Definire strutture dati come teorie

Per definire astrazioni di alto livello che consentano di concettualizzare le strutture dati (collezioni, stringhe, documenti, forme geometriche, ecc.) senza ridurle a un agglomerato di componenti (come avviene nella programmazione imperativa), sono necessarie tecniche differenti.

L'ideale sarebbe definire le strutture dati come teorie (in senso matematico), in modo da poter ragionare sulle loro proprietà. Una **teoria** consiste di:

- uno o più tipi di dati (che sono a loro volta teorie, ovvero una teoria viene definita a partire da altre teorie);
- delle operazioni su tali tipi;
- delle proprietà che definiscono le relazioni tra i valori e le operazioni.

Un aspetto importante delle teorie matematiche è che, in genere, *una teoria non descrive mutazioni*, cioè non fornisce meccanismi per modificare caratteristiche degli elementi della teoria. Infatti, le mutazioni possono compromettere la validità delle leggi/proprietà della teoria.

Il vantaggio dell'astrazione basata sulle teorie è la possibilità di ragionare in modo formale sui programmi, così come si fa appunto sulle teorie in ambito matematico, evitando in particolare le difficoltà di ragionamento legate alla semantica operazionale dei linguaggi imperativi (che, essendo basata sui cambiamenti di stato, rende difficile la scrittura delle equivalenze che tipicamente verrebbero usate in matematica per esprimere le proprietà di una teoria[1]).

1.1 Esempio: teoria dei polinomi

La teoria dei polinomi (in un'unica variabile) definisce delle relazioni sulla somma dei polinomi che consentono di affermare, ad esempio, che:

$$(ax + b) + (cx + d) = (a + c)x + (b + d)$$

[1] Esiste una forma alternativa di semantica, la *semantica denotazionale*, che risolve questo problema, ma in pratica la descrizione della semantica denotazionale dei linguaggi "reali" risulta troppo complessa.

Invece, essa non definisce operatori che consentano di modificare un coefficiente di un polinomio fissato. Se un tale operatore esistesse, si potrebbe ad esempio modificare un coefficiente di uno dei polinomi che compaiono a sinistra dell'identità mostrata prima, rendendola così non più valida.

Se si implementasse questa teoria in un linguaggio imperativo, l'impossibilità di modificare i coefficienti di un polinomio sarebbe responsabilità del programmatore, perché il linguaggio non lo garantisce, cioè ammette implementazioni che invece consentono la modifica dei coefficienti. Ad esempio, in Java, si potrebbero modellare i polinomi con una classe che contiene un array di coefficienti,

```
class Polynomial { double[] coefficients; }
```

e allora i coefficienti di un polinomio `p` potrebbero essere liberamente modificati:

```
Polynomial p = new Polynomial(/* ... */);
// ...
p.coefficients[0] = 12;
// ...
p.coefficients[0] = 33;
```

Si noti in particolare che il polinomio `p` è sempre lo stesso oggetto, nonostante i suoi coefficienti cambino, quindi una proprietà di `p` che è valida in un istante potrebbe non valere più dopo la modifica di un coefficiente.

1.2 Esempio: teoria delle stringhe

La teoria delle stringhe definisce un operatore $++$ di concatenazione delle stringhe, che ha ad esempio la proprietà di essere associativo:

$$(a ++ b) ++ c = a ++ (b ++ c)$$

Essa non definisce invece operatori per modificare un elemento di una stringa, a differenza di quanto può in generale avvenire nei linguaggi di programmazione imperativa.

In questo caso particolare, Java fa la cosa giusta (anche se per motivi di efficienza[2]): le stringhe sono oggetti non modificabili, in inglese **immutable**.

[2] Ad esempio, il fatto che le stringhe non siano modificabili consente di usare un singolo oggetto per rappresentare diverse istanze di una stessa stringa: se invece le stringhe fossero modificabili, ogni istanza dovrebbe essere rappresentata da un oggetto separato, in modo che eventuali modifiche di un'istanza non siano visibili anche sulle altre.

2 Programmazione funzionale

Un paradigma che fornisce gli strumenti per definire le strutture dati come teorie è la **programmazione funzionale** (**FP**, *Functional Programming*), che in particolare:

- si concentra sulla definizione di teorie con operatori espressi come funzioni (dove il termine funzione è inteso in senso matematico);
- evita le mutazioni;
- individua meccanismi (potenti) di astrazione sulle funzioni e di composizione delle funzioni.

Il termine *programmazione funzionale* ha due possibili significati:

- In "senso stretto", significa programmare utilizzando le funzioni come strumento di base, senza variabili mutable, assegnamenti, cicli e altre strutture di controllo imperative.
- In "senso ampio" significa utilizzare le funzioni come strumento centrale nell'attività di programmazione; in particolare, le funzioni possono essere generate durante il processo di computazione, utilizzate e composte. Questo può essere fatto anche in linguaggi non funzionali, ma i linguaggi funzionali forniscono appositi meccanismi per operare in questo modo.

Passando a considerare i linguaggi che implementano le due accezioni di questo paradigma, si ottengono due definizioni di **linguaggio di programmazione funzionale** (**FPL**, *Functional Programming Language*):

- in "senso stretto", è un linguaggio che non ammette variabili mutable, assegnamenti e strutture di controllo imperative;
- in "senso ampio", è un linguaggio che fornisce i meccanismi per costruire programmi eleganti che hanno le funzioni come elemento centrale dell'attività di programmazione.

In particolare, in un linguaggio di programmazione funzionale *le funzioni sono cittadini di prima classe*, cioè hanno il medesimo status che hanno gli altri valori nei linguaggi di programmazione:

- possono essere definite ovunque, anche all'interno di altre funzioni;
- possono essere passate come argomento alle funzioni e restituite come valori delle funzioni;
- esistono operatori che operano su di esse.

Dal punto di vista teorico, i linguaggi funzionali in senso stretto sono i più interessanti, in quanto, essendo totalmente privi di aspetti imperativi, si prestano particolarmente ai ragionamenti formali sui programmi. Questo un po' si perde nei linguaggi funzionali in senso ampio, anche se l'uso degli aspetti imperativi non è obbligatorio (tranne che, solitamente, per eseguire operazioni di input e output).