Programmazione Funzionale

Appunti delle Lezioni di Programmazione Funzionale

Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeXdisponibile al seguente link:
https://github.com/00Darxk/Programmazione-Funzionale/

Indice

1 Introduzione ad OCaml

La programmazione funzionale è un paradigma di programmazione estremamente potente, utilizza un metodo completamente diverso dagli altri paradigmi di programmazione. L'idea di base consiste nel considerare un programma una funzione, è un linguaggio di alto livello dichiarativo. Il programmatore infatti specifica che cosa deve essere calcolato, non come deve essere calcolato, a differenza di linguaggi imperativi. Il linguaggio di programmazione pià funzionale è Haskell, nel corso verrà usato il linguaggio Objective Caml, contiene aspetti di programmazione orientata agli oggetti. Objective Caml appartiene alla famiglia *Meta Language* ML sviluppata dall'INRIA in Francia dal 1984. Dietro un linguaggio c'è un modello di calcolo che determina le operazioni eseguibili ed anche lo stile di programmazione. Si basa sul lambda calcolo, un linguaggio Turing completo, estremamente semplice.

I tre costrutti fondamentali sono applicazioni, composizione e ricorsione. Su alcune distribuzioni di Linux l'interprete OCaml è preinstallato, ed è possibile interagirci digitando il comando `ocaml` su un terminale:

```
prompt> ocaml
OCaml version x.x.x
Enter #help;; for help.

#
```

Il compilatore legge un'espressione o una dichiarazione, terminata dalla sequenza `;;`, in seguito calcola il valore, deducendo i tipi delle variabili utilizzate, e restituisce la soluzione a schermo. Non c'è bisogno di dichiarazioni esplicite, OCaml può effettuare inferenze di tipo anche su espressioni estremamente complesse. È un linguaggio a tipizzazione statica, ogni tipo può essere determinato a tempo di compilazione.

I costrutti di base sono le espressioni, non sono comandi, hanno sempre un valore ed un tipo. Il calcolo procede valutando queste espressioni, semplificandole fino ad ottenere un'espressione non più semplificabile, cioè un valore.

La forma generale di una dichiarazione consiste dalla parola chiave `let` seguita dall'identificatore, a cui si assegna una certa espressione:

```
# let <identificatore> = <espressione>
```

Per specificare gli argomenti o parametri di una funzione, bisogna specificarli dopo il nome della funzione, per indicare che si tratta di una funzione ricorsiva bisogna utilizzare la parola chiave `rec`, dopo `let`:

```
# let rec <identificatore> <parametri> = <espressione>
```

Un ambiente è una collezione di legami tra variabili e valori, l'ambiente iniziale comprende tutte queste associazioni presenti nel modulo iniziale contenuto in `Stdlib`. Quando viene aggiunta ua nuova dichiarazione viene aggiunto un nuovo legame in cima a questo ambiente. Questo ambiente viene gestito come una pila, quindi dichiarazioni future sovrascrivono dichiarazioni precedenti,

poiché vengono accedute prima. Il valore delle variabili globali viene determinato a tempo di compilazione. Per cui è possibile modificare il valore di una variabile globale sovrascrivendola, ma altri oggetti possono comunque riferirsi alla vecchia definizione di questi valori, poiché si riferisce al tempo dove questa dichiarazione è stata inserita nell'ambiente.

```
# let one = 1;;
val one : int =  1
# let oneplus n = n + one;;
val oneplus : int → int = <fun>
# let one = 2;;
val one = 2
# oneplus 1;;
- : int = 2
```

Quando si effettua una definizione che non dipende dal tipo specificato dei suoi parametri, queste vengono sostituite da variabili di tipo `'a`, che rappresentano un tipo generico. Per indicare che si tratta di un tipo generico viene preceduto da un'apostrofo, convenzionalmente si usano le lettere greche α in questo caso. È possibile applicare questa dichiarazione a qualsiasi tipo.

Le funzioni sono oggetti di prima classe, hanno un proprio valore ed un tipo, quindi è possibile manipolarli, passando funzioni come argomenti ad altre funzioni.
Nelle espressioni di tipo si associa a destra:

```
int → int → int = int → (int → int)
```

Mentre nelle espressioni si associa a sinistra:

```
(func n) m = func n m
```

Per cui l'uso delle parentesi è essenziale per il buon funzionamento del codice.

Funzioni di ordine superiore sono dei costrutti che prendono come argomento o riportano come valore una funzione. In questo modo si possono realizzare semplicemente funzioni come la sommatoria, che prendono come argomento la funzione `f` di cui devono effettuare la somma:

```
# let rec sum f (lower, upper) =
    if lower > upper then 0
    else f lower + sum f (lower + 1, upper)
val sum : (int → int) → int * int → int = <fun>
```

La funzione `sum` può essere applicata anche soltanto al suo primo argomento, generando una funzione di prima classe. Invece di inserire una coppia si possono inserire due elementi, *currificando* la funzione, ovvero può funzionare anche con valutazione parziale, generando una funzione che si aspetta i rimanenti parametri, invece di generare un errore:

```
# let rec sum f lower upper =
    if lower > upper then 0
    else f lower + sum f (lower + 1) upper;;
val sum : (int → int) → int → int → int = <fun>
```

Una funzione non currificata può essere applicata parzialmente.
In generale f_c è la forma currificata di f se:

$$f : t_1 \times \cdots \times t_n \to t$$
$$f_c : t_1 \to (t_2 \to \cdots \to (t_n \to t) \cdots)$$

Currificando una funzione è possibile applicarla parzialmente.

Molte operazioni predefinite in OCaml sono in forma currificata. Le operazioni infisse predefinite sono in forma currificata. Per utilizzare un operatore in forma infissa si racchiude tra due parentesi tonde. Per definire degli operatori infissi si racchiudono tra parentesi nella loro definizione.

Per effettuare un'operazione di composizione il codominio della seconda funzione deve essere uguale al dominio della prima funzione. Restituisce una funzione f applicata su g.

Un tipo è l'insieme dei valori che può assumere, i tipi predefiniti sono i booleani, su cui sono definite le operazioni booleane `not`, `&&` e `||`.