

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 L^AT_EX disponibile al seguente link:

<https://github.com/00Darxk/Programmazione-Funzionale/>

Indice

1	Introduzione ad OCaml	1
----------	------------------------------	----------

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 più 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 una 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:

$$\begin{aligned} f &: t_1 \times \dots \times t_n \rightarrow t \\ f_c &: t_1 \rightarrow (t_2 \rightarrow \dots \rightarrow (t_n \rightarrow t) \dots) \end{aligned}$$

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:

- Booleani, su cui sono definite le operazioni booleane `not`, `&&` e `||`.
- Interi su cui sono definite le operazioni `+`, `-`, `*`, `/`, `mod`, `succ` e `pred`.
- Numeri reali `float`, su cui sono definite alcune delle stesse funzioni definite sugli interi, ma per mantenere l'inferenza di tipo vengono seguiti da un punto: `+. , -. , *. e /. .`. Oltre a queste sono definite le più importanti funzioni matematiche sui reali.
- Caratteri `char`, definiti sempre tra due apici ' ', si possono convertire in codice ASCII corrispondente e viceversa con `int_of_char` e `char_of_int`.
- Stringhe tra doppi apici " ", che possono essere concatenate con l'operatore `^`, si può accedere ai singoli caratteri con la notazione puntata specificando la posizione `. [i]`.

Se si utilizzano funzioni definite su interi su reali o viceversa, viene generato un errore, sono comunque presenti funzioni per convertire tra questi tipi, ma operazioni di casting vengono sconsigliate. Gli operatori di confronto sono definiti su qualsiasi tipo, eccetto che sulle funzioni. Nella notazione di OCaml il non uguale si rappresenta come `<>`. Si possono effettuare confronti tra tuple, controllando secondo l'ordine lessicografico partendo dalla prima componente.

Si possono realizzare istruzioni condizionali con il costrutto `if E then F else G`, dove E è un booleano, mentre F e G hanno lo stesso tipo, almeno un sottotipo in comune, e nei linguaggi ML deve essere possibile determinarlo a tempo di compilazione, sono linguaggi fortemente tipati.

Non è un costrutto di controllo come nei linguaggi imperativi, ma è un'espressione con un valore ed un tipo, il più generale tra F e G . Viene valutata in maniera pigra, se E è vera, non viene valutata G , mentre se è falsa non viene valutata F .

Le seguenti espressioni sono quindi equivalenti:

- `E && F`: `if E then F else false`.
- `E || F`: `if E then true else F`.

Per valutare le espressioni in linguaggi ML si utilizza la regola di calcolo *call by value*, la valutazione per valore, dove si calcola il valore dell'argomento prima di applicare la funzione, invece della valutazione per nome *call by name*, questa regola viene utilizzata solamente nelle espressioni condizionali ed operatori booleani. Se non si valutassero le espressioni booleane in modo pigro, ogni funzione ricorsiva che utilizza una funzione booleana per effettuare la ricorsione, valuterebbe all'infinito il passo ricorsivo, senza poter mai fermarsi.

Le coppie ordinate sono formate da due elementi divisi da una virgola tra parentesi tonde. Per rappresentare un costrutto di tipo si utilizza l'operatore `*` per indicarlo. Sulle coppie si utilizzano le funzioni `fst` e `snd` per restituire il primo ed il secondo elemento della coppia, queste sono polimorfe, ma se vengono usate su tuple da più di due elementi restituiscono un errore.