FUNZIONI

Abbiamo visto tre tipologie di forme sintattiche

(1) espressioni → oggetti sintattici con una nozione di riduzione

(2) valori → espressioni non ulteriormente riducibili

(3) definizioni → oggetti sintattici che vincolano valori a nomi

Introduciamo ora una ulteriore categoria: le funzioni

FUNZIONI ANONIME

$$\boxed{\text{fun } x \to e}$$

Corrisponde alla notazione matematica $\underbrace{x \mapsto e \quad \text{o} \quad \lambda x.e \quad \text{o} \quad \lambda x \to e}_{\lambda\text{-expressions}}$

Es. fun $x \to x+1$ è la funzione successore

Una funzione anonima è un'espressione

(fun x -> x+1) + (if 3 > 2 then 0 else 42) ...

Cosa succede se valutiamo fun x -> x+1 ?

⇒ -: int -> int = <fun>

leggendo da dx a sx, abbiamo che

1. Il valore di fun x -> x+1 è una funzione unprintable (<fun>)
2. Il tipo della funzione è int -> int

⇝ Funzioni sono valori

Prima di approfondire...

Es. $\underbrace{\text{fun } x \ y}_{\text{2 parametri}} \rightarrow x \underbrace{+.}_{} y$

↳ somma di float → Ocaml non fa type casting

$\xrightarrow{\text{eval}} \underbrace{\text{float} \rightarrow \text{float}}_{\text{2 argomenti}} \rightarrow \text{float} = \text{<fun>}$

ordine superiore

int { + * − / : 0 1 :

+. *. −. /. : } float
0.0
1.0
:

fun x y z → e avrà tipo

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_{exp}$

Def. Una funzione anonima è una espressione della forma

$$\text{fun } x \to e$$

SINTASSI $\quad \text{fun } \underbrace{x}_{\text{variabile / parametro formale}} \to e$] espressione / corpo della funzione

La variabile x è legata in e

- $\text{fun } x \to \boxed{\;\cdot\;\cdot\;\cdot\;\cdot\;\cdot\;}$

  scope

- $\text{fun } x \to e \cong_\alpha \text{fun } y \to e[y/x]$

STATICA. Le funzioni hanno tipo freccia

$$T, S ::= int \mid float \mid bool \mid \dots \mid T \to S$$

Intuizione: una funzione ha tipo $T \to S$ se prende in input qualcosa di tipo T, e restituisce qualcosa di tipo S

Regola: $\text{fun } x \to t : T \to S$ se

t ha tipo S assumendo che x abbia tipo T

$$\frac{\Gamma, x : T \vdash t : S}{\Gamma \vdash \text{fun } x \to t : T \to S}$$

DINAMICA

$\text{fun } x \to t \Rightarrow \text{fun } x \to t$

una funzione è un valore!

Esattamente come lo sono true, 0, 118, ...

→ Quando si esegue una funzione

$\text{fun } x \to t$

~~non~~ viene valutato il corpo della funzione

$\text{fun } x \to x+0 \not\Rightarrow \text{fun } x \to x$

anche se $x+0 \Rightarrow x$

CONSEGUENZE

- Possiamo scrivere funzioni che restituiscono funzioni

$$\text{fun } x \to \underbrace{(\text{fun } y \to x + y)}_{\text{espressione}}$$

Zucchero sintattico : $\text{fun } \underbrace{x \quad y}_{} \to x + y$

2 parametri, ma sintassi forza 1 parametro

$\text{fun } x \to t$

- Statica di: $\text{fun } x \to (\text{fun } y \to x + y)$

$$int \to (int \to int)$$

Questo processo si chiama Currying (dal nome del logico Haskell Curry).

Teorema. Una funzione in n-parametri:

$$\text{fun } x_1 \dots x_n \to t$$

è simulata (vedremo dopo) da una funzione di ordine superiore in 1 parametro

$$\text{fun } x_1 \to (\text{fun } x_2 \to \dots (\text{fun } x_n \to t) \dots)$$

APPLICAZIONE DI FUNZIONI

D. Come possiamo usare una funzione? ⇝ Applicazione (passaggio di argomento)

Def. Date espressioni t, s, l'applicazione di t ad s è l'espressione

$$t \ s$$

NB. Non si usano parentesi $t(s)$, $(t)s$ ...

Ex. $(\text{fun } x \to x+1)\ 5$

$f \ 2$

$\vdots$

SINTASSI $t, s ::= \ldots \mid t\ s$

STATICA Se t è un'espressione di tipo $S \to T$ e s è un'espressione di tipo S, allora $t\ s$ ha tipo T

$$\frac{t : S \to T \qquad s : S}{t\ s : T}$$

DINAMICA Per valutare $t\ s$ si fa:

1. Si valuta t, ottenendo un valore necessariamente (se tipaggio ok) $\text{fun } x \to t_0$
2. Si valuta s, ottenendo un valore v
3. Si valuta $t_0[v/x]$

Simbolicamente

$$\frac{t \Rightarrow \text{fun } x \to t_0 \qquad s \Rightarrow v \qquad t_0[v/x] \Rightarrow v_0}{t\,s \Rightarrow v_0}$$

Ex.

$$\begin{aligned}
&(\text{fun } x \to x+1)\ \underline{(2+3)} \\
\rightsquigarrow\ &\underline{(\text{fun } x \to x+1)\ 5} \\
\rightsquigarrow\ &\underline{5+1} \\
\rightsquigarrow\ &6
\end{aligned}$$

REGOLA CRUCIALE (β-REGOLA)

corpo della funzione

$$(\text{fun}\ \underbrace{x}_{\text{parametro formale}} \to t)\ \underbrace{v}_{\text{parametro attuale}} \rightsquigarrow \underbrace{t[v/x]}_{\text{sostituzione parametro attuale per parametro formale nel corpo della funzione}}$$

parametro formale

parametro attuale

sostituzione parametro attuale per parametro formale nel corpo della funzione

➡ Essenza del modello di: computazione per sostituzione

Programmazione di Ordine Superiore

Un linguaggio di programmazione è di ordine superiore
dati e programmi del linguaggio coincidono

⇝ Un programma può prendere in input e restituire
in output un altro programma

- $\text{fun } x \to \underbrace{(\text{fun } y \to x + y)}_{\text{output è funzione}}$

$$(\text{fun } x \to (\text{fun } y \to x + y))\ 5 \Rightarrow \text{fun } y \to 5 + y$$

- $\text{fun } f \to (\text{fun } x \to f\ (f\ x))$ } doppia iterazione

$$(\text{fun } f \to \underbrace{(\text{fun } x \to f\,(f\,x))}_{\text{funzione in output}})\ \underbrace{(\text{fun } x \to x+1)}_{\text{funzione in input}}$$

$$\cong_\alpha (\text{fun } f \to (\text{fun } y \to f\,(f\,y)))\ (\text{fun } x \to x+1)$$

$$\overset{\beta}{\rightsquigarrow} \underbrace{\text{fun } y \to (\text{fun } x \to x+1)\ ((\text{fun } x \to x+1)\ y)}_{\text{questo è un valore}}$$

- Programmazione di ordine superiore è un potentissimo meccanismo di astrazione

- Anima dell'informatica

  - Program $\cong$ Data
  - Universalità

- Ma anche comportamenti strani

$$\boxed{\begin{array}{l} \text{let } \Delta = \text{fun } x \to x\ x \\ \text{in } \Delta\ \Delta \end{array}}$$

Questo programma si chiama Ω

DEFINIZIONI DI FUNZIONI

Spesso conviene dare un nome alle funzioni anonime

```
let add = fun x y -> x + y ;;
```

OCaml offre zucchero sintattico

```
let add x y = x + y ;;
```

Definizione: non espressione

eventualmente con annotazioni di tipo

```
let add (x : int) (y : int) : int = x + y
```

zucchero sintattico $\boxed{\text{let } f\ x = t}$ → corpo funzione : x è legata in t

$\underbrace{\text{nome funzione} \quad \text{parametro formale}}_{\text{identificatori}}$

DINAMICA (CON AMBIENTE)

$$\frac{\eta[f \mapsto \mathrm{FUN}(x,t)] \mid s \Rightarrow v}{\eta \mid \mathrm{let}\ f\ x = t\ \mathrm{in}\ s \Rightarrow v}$$

NON ESATTAMENTE CORRETTA, MA OK PER ORA

$$\frac{\eta \mid t \Rightarrow \mathrm{fun}\ x \to t' \qquad \eta \mid s \Rightarrow v \qquad \eta \mid t'[v/x] \Rightarrow w}{\eta \mid t\ s \Rightarrow w}$$

$$\frac{\eta \mid t \Rightarrow f \qquad \eta(f) = f \mapsto \mathrm{FUN}(x,t) \qquad \eta \mid s \Rightarrow v \qquad \eta[x \mapsto v] \mid t \Rightarrow w}{\eta \mid t\ s \Rightarrow w}$$

Domanda. Perché parliamo di programmazione funzionale?

... varie categorie sintattiche

- espressioni
- definizioni
- definizioni di funzione

→ espressione

... varie espressioni

- 0, 1, 2, ...
- let x = t in s
- fun x → t

→ funzione

Ex. let $x = t$ in s

è zucchero sintattico per

$(\text{fun } x \to s)\ t$

$$\frac{t \Rightarrow v \qquad s[v/x] \Rightarrow w}{\text{let } x = t \text{ in } s \Rightarrow w}$$

$$\frac{\text{fun } x \to s \Rightarrow \text{fun } x \to s \qquad t \Rightarrow v \qquad s[v/x] \Rightarrow w}{(\text{fun } x \to s)\ t \Rightarrow w}$$

stesso risultato

$\rightarrow$ let $x = t$ in $s \cong_\alpha$ let $y = t$ in s

deriva da $\underline{\text{fun } x \to s \cong_\alpha \text{fun } y \to s[y/x]}$ (α-regola)

Piccolo nucleo concettuale

$$\underbrace{\text{fun } x \to t}_{\text{astrazione}} \qquad \underbrace{t\ s}_{\text{applicazione}}$$

$$\text{fun } x \to t \ \cong_\alpha \ \text{fun } y \to t[y/x] \qquad \alpha\text{-regola}$$

- scoping
- renaming

⋮

$$(\text{fun } x \to t)\ v \ \to_\beta \ t[v/x] \qquad \beta\text{-regola}$$

- computazione

λ-Calcolo

Ex.

```
let f x y = x - y;;
f 3 2 ;;
```

nested let-exp ⇝

```
let f x y = x - y
in f 3 2
```

fun. def ⇝

```
let f = fun x -> (fun y -> x - y)
in f 3 2
```

let sotto fun ⇝

```
(fun g -> g 3 2) (fun x -> (fun y -> x - y))
```

Ora possiamo applicare la <u>β-regola</u>

FUNZIONI RICORSIVE

let rec f x = t } ricorsione deve essere esplicitata

Ex.
```
let rec fact n =                    (* PRE: [n ≥ 0] *)
    if n = 0 then 1
    else n * fact (n-1)
```
$\underbrace{\texttt{fact (n-1)}}$ ↳ notare differenza con fact n −1

Eseguiamo ⇝ val fact : int → int = <fun>

Ex.
```
let rec pow x y =
    if y = 0 then 1
    else x * pow x (y-1)
```

SINTASSI

$$\text{let rec } \underbrace{f\ x}_{\text{identificatori / parametri formali}} = \underbrace{t}_{\text{corpo della funzione}} \text{ in } \underbrace{s}_{\text{continuazione}} \quad \cdots\cdots\!> \text{ è una } \underline{\text{espressione}}$$

zucchero sintattico

$$\text{let rec } f\ x = t \quad \cdots\cdots\!> \underline{\text{definizione}} \text{ di funzione}$$

NB. f può comparire in t

<u>STATICA</u>. Per inferire che

let rec $f\ x = t$ ha tipo $T \to S$

dobbiamo far vedere che

t ha tipo S

<u>assumendo</u> che

x ha tipo T

f ha tipo $T \to S$

$$\boxed{\dfrac{\Gamma,\ x : T,\ f : T \to S \vdash t : S}{\Gamma \vdash \text{let rec } f\ x = t : T \to S}}$$

__DINAMICA__

$\text{let rec } f\ x = t \text{ in } s$

assegna al nome f la funzione che mappa x in t

$$\frac{\eta[f \mapsto FUN\_REC(x,t)] \mid s \Longrightarrow w}{\eta \mid \text{let rec } f\ x = t \text{ in } s \Longrightarrow w}$$

non proprio corretta, ma per ora ok

DINAMICA APPLICAZIONE

$$\frac{\eta \mid t \Rightarrow f \qquad \eta(f) = \text{FUN\_REC}(x, t) \qquad \eta \mid s \Rightarrow v \qquad \eta[x \mapsto v] \mid t \Rightarrow w}{\eta \mid t\,s \Rightarrow w}$$

<u>DOMANDA</u>. Serve davvero tutto ciò?
Possiamo ridurre tutto ad espressione?

Sì, ma lo vedremo più avanti...

Per ora pensiamo a funzioni anonime ricorsive

$$\boxed{\text{fix } f\ x \to t}$$ → <u>NON</u> è sintassi Ocaml

con <u>β-regola</u>

$$(\text{fix } f\ x \to t)\ v \to_\beta t\left[{}^{v}/_{x},\ {}^{\text{fix } f\ x \to t}/_{f}\right]$$

Ex. Factorial

$$\underbrace{\text{fix } f\ n \to \text{if } n = 0 \text{ then } 1 \text{ else } n * f\,(n-1)}_{\text{fact}}$$

$$\text{fact } 2 \to_\beta \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * \text{fact } (2-1)$$

$$\to 2 * \text{fact } (2-1)$$

$$\to 2 * \text{fact } 1$$

$$\to 2 * (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * \text{fact } (1-1))$$

$$\vdots$$

$$\to 2 * 1$$

$$\to 2$$

Polimorfismo

Consideriamo la funzione identitá

$\text{let id } x = x \qquad (\text{let id} = \text{fun } x \to x \text{ in } \dots)$

Qual è il tipo di id?

$$
\left.
\begin{aligned}
\text{id}:\ & \text{int} \to \text{int} \\
& \text{bool} \to \text{bool} \\
& (\text{int} \to \text{bool}) \to (\text{int} \to \text{bool}) \\
& \vdots
\end{aligned}
\right\}
$$

polymorphic type

$\forall \alpha . \alpha \to \alpha$

variabile di tipo

Un'espressione di tipo $\forall \alpha . \alpha$ può essere istanziata con ogni tipo

In OCaml la grammatica dei tipi comprende variabili di tipo

$$T, S ::= \underbrace{int \mid bool \mid float}_{\text{tipi base}} \mid \ldots \mid \underbrace{\alpha \mid \beta}_{\text{variabili di tipo}} \mid \ldots \mid \underbrace{T \to S}_{\text{costruttori di tipo}}$$

Le variabili di tipo possono assumere qualsiasi "valore"

$id : \alpha \to \alpha$ (OCaml: `'a -> 'a`
$\rightsquigarrow$ viene omesso il $\forall \alpha.$)

Quando si passa ad id una espressione $E : T$, allora α viene istanziata a T, e quindi id ad $T \to T$

$\underbrace{\text{Poly}}_{\text{molte}}\underbrace{\text{morphism}}_{\text{forma}}$ $\rightsquigarrow$ scrivere programmi / funzioni che possono agire sul massimo numero di argomenti

$$\left.\begin{array}{l} id:\ int \to int \\ id:\ bool \to bool \\ id:\ (int \to bool) \\ \qquad \to (int \to bool) \\ \quad \vdots \end{array}\right\} \alpha \to \alpha$$

Potente principio di astrazione

- Generics (Java) List<T>
- Templates (C++)

$\vdots$

Quando OCaml fa type inference, inferisce il tipo più generico possibile usando il polimorfismo