FP

Lez 3:

Ricorsione

RECAP

p.s. Che cos'è una funzione?

- Nome “funzione” introdotto da *Leibinz* nel 1673, in ambito analitico, poi chiarito da *Bernoulli* e *Euler* come “ogni espressione fatta di variabili e costanti”
- *Hardy* 1908: una funzione è una relazione tra x ed y che "to some values of x at any rate correspond values of y."
- Bourbaki 1939: “"Let E and F be two sets, which may or may not be distinct. A relation between a variable element x of E and a variable element y of F is called a *functional relation* in y if, **for all** $x \in E$, **there exists a unique** $y \in F$ which is in the given relation with x.”
- Queste sono def *estensionali*. Alternativamente: λ-calcolo

Espressioni

- Modello di computazione = *valutazione di espressioni*
- Ogni expr:
  - ha o non ha un **tipo**
  - ha o non ha un **valore** (non-terminazione/run time error)
  - può generare un **effetto** (I/O, eccezioni etc)

Tipi

- Tipi = valori + operazioni
- "exp : ty" (espressione "exp" ha tipo "ty") è una *predizione* (statica) della forma del valore di *exp*, se converge
- "exp : ty" – *asserzione* (detta anche *giudizio*) è **valida** se posso dimostrare (con una *derivazione di tipo*) che *exp* ha effettivamente tipo *ty*
  - Giudizio (3 + 4) : int è **valido**
  - (3 + true) : bool **non è valido**

Let

- Lego un valore "val" a una variabile "id", notazione "id →val" ("*binding*")
- Un “*enviroment*” (ambiente) è un insieme di binding
  - Più esattamente, enviroment é una **funzione parziale finita** tra *id* e *val*
  - Rappresentata conme una lista *snoc* (che si estende a destra) con questa BNF
    - η ::= . | η , id → val

Let globale

- `let id = exp`

```
let x = 2;
let y = true;;
let f = fun k -> (x,y,k);;
```

- generano enviroment

  ., x → 2, y → true, f → fun k -> (x,y,k)

- Vi è anche un ambiente predefinito per costanti etc.

```
System.Math.PI;;
val it : float = 3.141592654
```

Let locale

- `let id = exp1 in exp2`

  binding *id* → *exp1* perso dopo valutazione di *exp2*
- F#: sintassi *light* (indentation sensitive) vs verbose

```
let x3 =                                    // globale
   let y = sin 4.0
   let z =  cos 1.0
   y + z;;

let x3 = let y = sin 4.0 in let z = cos 1.0
in y + z
```

Code

Mantra sul let

- Espressione *let* **non** è assegnamento
- Le variabili non variano, sono cioè per default **immutabili**
- Le variabili hanno un ambito (“*scope*”) in cui hanno senso
- Se ri-lego un valore *val* a variable *id*, vale il legame più recente (“*shadowing”)*

Tipi e valori revisited

- In presenza di variabili, dobbiamo generalizzare la nozione di derivazione di tipo attraverso la nozione di enviroment statico o *contesto*
  - $\Gamma ::= . \mid \Gamma , x : \tau$
  - $\Gamma \models exp : ty$
- Similmente vi è un giudizio per la *valutazione* di expressioni
  - $\eta \models exp \gg val$
- Vedremo le regole per questi giudizi più avanti

Pattern matching

- Il metodo standard per analizzare dati in FP
- PM su expressioni con tipi primitivi, es interi:

```
let f n =
 match n with
    |0 -> e1
    | m -> e2
```

- PM su espressioni complesse come tuple:

```
let And (x,y)  =
  match (x,y) with
    | (true, true) -> true
    | _ -> false
```

Pattern matching cont.

- PM è cronologico – ordine conta
- PM dovrebbe essere:
  - *Esaustivo*: pattern coprono ogni possibile forma
  - *Disgiunto*: non vi siano pattern con overlap
    - Interprete segnala un warning
- In un ramo di PM `p → e`, le variabili che occorrono in `p` sono vincolate e come tali
  - Il loro nome non conta (α-renaming)
  - Entrano a far parte dell'enviroment locale

PM & let

- Più generalmente una espressione *let* prende non solo un *id*, ma un pattern, di cui *id* è una istanza
  - `let pat = exp1 in exp2`
- Spesso *let* è definito in termini di *match*

```
let p = (1,2)
let (x,y) = p in x + y
match p with (x,y) -> x + y
```

- `pat::= k | x | _ |(pat1,pat2)|` …

RICORSIONE

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$
\begin{array}{rcll}
0! & = & 1 & (i) \\
n! & = & n \cdot (n-1)!, \quad \text{for } n > 0 & (ii)
\end{array}
$$

Computation:

$$
\begin{array}{rll}
 & 3! & \\
= & 3 \cdot (3-1)! & (ii) \\
= & 3 \cdot 2 \cdot (2-1)! & (ii) \\
= & 3 \cdot 2 \cdot 1 \cdot (1-1)! & (ii) \\
= & 3 \cdot 2 \cdot 1 \cdot 1 & (i) \\
= & 6 &
\end{array}
$$

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$
\begin{array}{rcll}
0! & = & 1 & (i) \\
n! & = & n \cdot (n-1)!, \quad \text{for } n > 0 & (ii)
\end{array}
$$

Computation:

$$
\begin{array}{rll}
 & 3! & \\
= & 3 \cdot (3-1)! & (ii) \\
= & 3 \cdot 2 \cdot (2-1)! & (ii) \\
= & 3 \cdot 2 \cdot 1 \cdot (1-1)! & (ii) \\
= & 3 \cdot 2 \cdot 1 \cdot 1 & (i) \\
= & 6 &
\end{array}
$$

Recursive declaration. Example *n*!

Function declaration:

```
let rec fact = function
  | 0 -> 1                   (* i *)
  | n -> n * fact(n-1);;   (* ii *)
val fact : int -> int
```

Evaluation:

$$\begin{array}{llll} & \mathtt{fact}(3) & & \\ \rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [n \mapsto 3] \\ \rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [n \mapsto 2] \\ \rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & [n \mapsto 1] \\ \rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [n \mapsto 0] \\ \rightsquigarrow & 6 & & \end{array}$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursive declaration. Example n!

Function declaration:

```
let rec fact = function
  | 0 -> 1                  (* i *)
  | n -> n * fact(n-1);;   (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllr}
 & \texttt{fact}(3) & & \\
\leadsto & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\leadsto & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\leadsto & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\leadsto & 3 * 2 * 1 * 1 & (i) & [\texttt{n} \mapsto 0] \\
\leadsto & 6 & &
\end{array}
$$

$e_1 \leadsto e_2$ reads: e_1 evaluates to e_2

Recursive declaration. Example n!

Function declaration:

```
let rec fact = function
  | 0 -> 1                  (* i *)
  | n -> n * fact(n-1);;   (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllr}
 & \mathtt{fact}(3) & & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & [\mathtt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\mathtt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursive declaration. Example $n!$

Function declaration:

```
let rec fact = function
  | 0 -> 1                    (* i *)
  | n -> n * fact(n-1);;      (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllll}
 & \mathtt{fact}(3) & & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & [\mathtt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\mathtt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursive declaration. Example n!

Function declaration:

```
let rec fact = function
  | 0 -> 1                      (* i *)
  | n -> n * fact(n-1);;   (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllll}
 & \mathtt{fact}(3) & & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & [\mathtt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\mathtt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursive declaration. Example n!

Function declaration:

```
let rec fact = function
  | 0 -> 1                    (* i *)
  | n -> n * fact(n-1);;      (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
 & \mathtt{fact}(3) & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) \quad [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) \quad [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) \quad [\mathtt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) \quad [\mathtt{n} \mapsto 0] \\
\rightsquigarrow & 6 &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$
\begin{array}{rcll}
x^0 & = & 1 & (1) \\
x^n & = & x \cdot x^{n-1}, \quad \text{for } n > 0 & (2)
\end{array}
$$

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                    (* 1 *)
  | (x,n) -> x * power(x,n-1)       (* 2 *)
```

Patterns:

`(_,0)` matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

`(x,n)` matches any pair (u, i) yielding the bindings

$$\mathtt{x} \mapsto u, \mathtt{n} \mapsto i$$

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$
\begin{aligned}
x^0 &= 1 && (1)\\
x^n &= x \cdot x^{n-1}, \quad \text{for } n > 0 && (2)
\end{aligned}
$$

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                 (* 1 *)
  | (x,n) -> x * power(x,n-1)    (* 2 *)
```

Patterns:

`(_, 0)` matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

`(x, n)` matches any pair (u, i) yielding the bindings

$$\mathtt{x} \mapsto u, \mathtt{n} \mapsto i$$

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$x^0 = 1 \qquad (1)$$
$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \qquad (2)$$

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                 (* 1 *)
  | (x,n) -> x * power(x,n-1)    (* 2 *)
```

Patterns:

$(\_, 0)$ matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

(x, n) matches any pair (u, i) yielding the bindings

$$x \mapsto u, n \mapsto i$$

Evaluation. Example: `power(4.0, 2)`

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                 (* 1 *)
  | (x,n) -> x * power(x,n-1)   (* 2 *)
```

Evaluation:

$$
\begin{array}{lll}
 & \mathtt{power}(4.0, 2) & \\
\rightsquigarrow & 4.0 * \mathtt{power}(4.0, 2-1) & \text{Clause 2, } [\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 2] \\
\rightsquigarrow & 4.0 * \mathtt{power}(4.0, 1) & \\
\rightsquigarrow & 4.0 * (4.0 * \mathtt{power}(4.0, 1-1)) & \text{Clause 2, } [\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 1] \\
\rightsquigarrow & 4.0 * (4.0 * \mathtt{power}(4.0, 0)) & \\
\rightsquigarrow & 4.0 * (4.0 * 1) & \text{Clause 1} \\
\rightsquigarrow & 16.0 &
\end{array}
$$

Ricorsione

code