FP

Lez 3:

Recap & Ricorsione

RECAP

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 corresponds 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 ∈ E, **there exists a unique** y ∈ F which is in the given relation with x.”
- Queste sono def *estensionali*. Alternativamente: λ-calcolo

Espressioni

- Modello di computazione = *valutazione di espressioni*
- Ogni *espressione*:
  - 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

- Tipo = insieme di valori + operazioni
- *Asserzione* "**exp** : **ty**" (espressione "exp" ha tipo "ty") è una *predizione* (statica) della forma del valore di *exp*, se converge
- "exp : ty" (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**
  - Giudizio (3 + true) : bool **non è valido**

Let

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

Let globale

- Esempi:

```
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)

  - Scritto nel libro (1.7) come [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 =                                  // x3 globale
  let y = sin 4.0
  let z =  cos 1.0
  y + z;;
```

- Equivalente a (sintassi *verbose*)

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

Mantra sul let

- *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")*

Code

Lez03.fsx

Tipi e valori revisited

- In presenza di variabili, dobbiamo generalizzare la nozione di derivazione di tipo attraverso la nozione di enviroment statico o ***contesto***
  - $\Gamma ::= . \mid \Gamma , x : \tau$ // una lista di coppie id,type
  - $\Gamma \models exp : ty$ // giudizio *exp* ha tipo *ty* in Γ
- Similmente vi è un giudizio per la *valutazione* di expressioni
  - $\eta \models exp \Rightarrow val$ // η è un enviroment
- 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), ma meglio siano differenti
  - Entrano a far parte dell'enviroment locale
  - Non possono avere ripetizioni, e.g. (x,x) non è valido pattern

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`
- **let** è definito in termini di **match**

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

- Grammatica dei pattern

```
::= k | x | _ |(pat1,pat2)| …
```

Back to code