Programmazione funzionale

- Costrutti di controllo: principalmente ricorsione
- Sistema dei tipi:
  - inferenza di tipo, polimorfismo
  - Tipi di dato
    - Lambda calcolo, un unico dominio: le funzioni (no tipi)
    - Lisp: atomi e liste (no tipi)
    - L. funzionali evoluti alcuni domini semplici(int, float, char, boolean), e strutturati liste e ennuple, tipi unione, tipi definiti dall'utente.

Caratteristiche di Caml

- Scaricabile: http://caml.inria.fr/ meglio ocaml
- è un linguaggio funzionale implementazione del λ-calcolo (Church A. 1932): usa la definizione e l'applicazione di funzioni come concetti essenziali.
  - Integrità referenziale
  - Ordine superiore: le funzioni sono valori del linguaggio (gli unici nel λ-calcolo puro).

Caratteristiche del sistema

- È un sistema interattivo: (di base ma esistono anche compilatori)
  - Si entra in un ambiente in cui il sistema stampa un prompt (#) e resta in attesa di una richiesta da parte dell'utente.
  - L'utente può immettere una frase in Caml (quali le frasi corrette?).
  - Il sistema calcola, stampa il risultato e un nuovo prompt.

Funzioni

In matematica:

$$f(x)=x+x$$

definizione della funzione *f* con parametro *x*.

$$f(2)$$

applicazione di *f* al valore 2.

In Caml

```
function x -> x+x
```

definisce una funzione **anonima**

```
(function x -> x+x) 3
```

è l'applicazione della funzione anonima ad un valore.

computazione per riduzione:

(function x -> x+x) 3 *riduceA* 3+3 *riduceA* 6

Costrutto let in

Introduce nomi.

```
let f=function x -> x+x in f 3;;
```

Sintassi: **let** *ide=exp* **in** *exp*

– L'identificatore esiste solo durante la valutazione dell'espressione. *riduceA*

```
(function x -> x+x) 3
```

rimpiazzamento di f nella parte destra con l'espressione che definisce f (a destra dell'uguale)

Integrità referenziale

- Integrità referenziale:
  - le espressioni denotano valori
  - Il significato di tutte le espressioni (in Caml e in tutti i linguaggi funzionali) è SOLO il valore. Qualunque metodo venga utilizzato per ottenerlo non produce altri effetti (non vero per gli altri linguaggi di programmazione)
  - Il valore (o forma più semplice di una espressione) è in realtà una rappresentazione di tale valore (rappresentazione dell'intero 10).

Riduzione di espressioni

- Valutazione di un'espressione, è un processo di riduzione o semplificazione (espressione canonica o in forma normale).
  - (2+3*4) → 2+12 → 14.
  - double (3+2) → double (5) → 5+5 → 10
- A volte ci sono più sequenze di riduzione possibili, es:
  - double (3+2) → (3+2)+ (3+2) → 5+(3+2) → 5+5 → 10

Riduzione di espressioni

- Un valore `e un'espressione non riducibile anche detta forma canonica o in forma normale. Es:
  - 14
  - function x -> x+1
- Un redex `e un'espressione che può essere ridotta. Es:
  - 2+3
  - (function x -> x+1) 4
- Strategie di valutazione, per selezionare il redex da ridurre se ne esiste più di uno nell'espressione.
  - nome, valore, lazy.

I tipi funzione

- Funzioni: type -> type

  Esempi di definizione di funzione anonime:

  - function x -> x+1 (succ)
  - function x -> x+x (double)
  - function x -> x*x (x^2)

  Esempi di definizioni con let:

- **let** succ = function x -> x+1 **in**...

  oppure let succ x = x+1 **in**... (abbreviazione sintattica)

- **let** double =function x -> x+x **in**...

  oppure **let** double x = x+x **in**...

Funzione a più argomenti

funzioni che calcolano funzioni:

```
function x -> function y -> x+y
let sum = function x -> function y -> x+y;;
```

sum 3 calcola: function y -> 3+y

quindi sum 3 4 calcola 3+4 cioè 7.

```
let sum x y = x+y;;
```
(abbreviazione sintattica)

In questo modo ho l'applicazione parziale

Funzione currizzate

Le funzioni cosi` definite si dicono currizzate.

In alternativa si possono definere funzioni che hanno come dominio il prodotto cartesiano dei domini degli argomenti:

Es: `sum x y=x+y`

oppure `sumpair(x,y)=x+y`

È possibile definire delle funzioni per passare da un modo all'altro:

```
curry f x y=f(x,y)          curry sumpair 3 4
uncurry f (x,y)= f x y      uncurry sum (3,4)
```

Costrutto let

- Dal momento che sarebbe molto noioso dover sempre ridefinire tutte le funzioni da utilizzare esiste un ambiente globale che contiene le definizioni delle funzioni. Tale ambiente è costruito con il costrutto **let.**

  let double x=x+x;;

  double 3;;

- Gli identificatori (nomi di variabili) sono associati a valori (funzioni), mai modificabili.

I tipi in Caml

- Ogni espressione ha un tipo (linguaggio fortemente tipato).
- I tipi delle espressioni vengono calcolati dal sistema (inferenza dei tipi). Non è necessario dichiarare i tipi.
- I tipi primitivi sono:
  - int: 0, 1,2,...-1,-2.. operazioni + , - , * , ecc
  - float: 1.2, 3.2, -4.5,... operazioni +. , -. , ecc.
  - boolean: true, false, operazioni & , or , not,
  - char, ‘a’,’b’,..
  - string: “Anna”,.. operazioni ^,s.[n], length

I tipi in Caml

- Tipi complessi:
  - Predefiniti:
    - Funzioni type -> type
    - Prodotto cartesiano: (type * type *..) (record)
    - Liste: type list
  - Definibili dall'utente:
    - Tipo unione con costruttori di tipo: per valori eterogenei, ogni componente dell'unione ha un costrutore.

Inferenza dei tipi e polimorfismo

È il sistema che scopre (inferisce) il tipo dei valori (e dei nomi) che vengono calcolati.

Es. `sum:int -> int -> int  double: int -> int`

Ma quale è il tipo di const:

```
let const x y = x;;
```

e quello di `curry f x y=f(x,y)`

e quello di `uncurry f (x,y)= f x y`

Variabili di tipo (‘a ‘b ‘c ecc.) e inferenza

Inferenza dei tipi

compose è una funzione che ha 3 argomenti cosi` definita:

let rec compose f g x = f(g(x));;

- cosa calcola? inferiamo il tipo.
  1. f:'a
  2. g:'b:
  3. x:'c
  4. compose :'a->'b->'c->'d
- da g(x) abbiamo che 'b='c->'e
- da f(g(x) abbiamo che a'='e->'d
- sostituendo in 1,2 e 4 abbiamo:
  - f:'e->'d e g: 'c->'e
  - compose:('e->'d)->('c->'e)->c'->'d

Inferenza dei tipi

```
let rec const x y =x
let rec foo f g x y =if (f x) then (g y) else (f y)
let rec subst f g x =f x (g x)
subst : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c =
    <fun>
```

uncurry compose

compose uncurry curry

compose curry uncurry

Tipi record

- Sono ennuple con nomi, es:
- type razionale={num:int; den:int}
- Si costruiscono:

  let r ={num:4; den:5}
- Si accedono con la sintassi usuale dei record

  r.num

Tipi unione (o somma)

- Tipi che raccolgono valori eterogenei:
  - hanno costruttori senza o con argomenti. Es.
    - type coloreB= Rosso | Giallo | Blu;;

      definizione del tipo coloreB che in questo caso ha 3 valori. Rosso, Giallo e Blu che sono costruttori (di valori di tipo colore).
  - I costruttori di tipo iniziano sempre con la maiuscola.
  - funzioni con il match:

```
let convert x = match x with
Rosso -> 1
Giallo -> 2
Blu -> 3;;
```

Tipi unione (o somma)

- Tipi unione con costruttori con argomenti:
  - type alberoBin = Foglia of string
    | Node of string * alberoBin * alberoBin
  - definizione del tipo alberoBin per rappresentare alberi con 2 figli:
  - let a=Node (“a”,Node(“b”,(Foglia “c”),(Foglia “d”)), Node(“e”,(Foglia“f”),(Foglia“g”)));;

Tipi polimorfi

- Tipi polimorfi definiti dall'utente parametrici rispetto a uno o più tipi:
  - type 'a alberoBin = Foglia 'a
    | Nodo of 'a * 'a alberoBin * 'a alberoBin
  - definizione del tipo alberoBin per rappresentare alberi con 2 figli:
  - let a=Nodo ("a",Nodo("b",(Foglia "c"),(Foglia "d")), Nodo("e",(Leaf "f"),(Leaf "g")));;
  - La variabile di tipo 'a è istanziata con il tipo string.
  - il tipo 'a mu per la memoria

Oltre il funzionale puro: Eccezioni

type exn = ... | Division_by zero | Failure **of** string ...

è un tipo predefinito che però può essere esteso:

exception MiaException **of** float

come usuali tipi e valori abbiamo

let e=MiaException 5.8

let e=Failure "errore"

per sollevare un'eccezione: **raise** (MiaException 5.6)

raise ha tipo 'a.

Oltre il funzionale puro:
Eccezioni

```
try exp with p1 -> e1
            | p2 -> e2
            ...
            | pn -> en
```

`exp e1,...en` sono espressioni tutte dello stesso tipo `p1...pn` sono pattern di tipo `exn`.

Il significato è intuibile si calcola exp se non fallisce l'intero costrutto calcola il risultato di exp. Altrimenti l'eccezione generata può essere catturata da uno dei pattern in tal caso il costrutto calcola il valore dell'espressione corrispondente.

Oltre il funzionale puro:
Eccezioni

```
let hd xs = match xs with
[] -> raise (Failure "hd")
| x::ys -> x;;
```

Esiste predefinita la funzione failwith:

```
let failwith s = raise (Failure s);;
```

```
let hd xs = match xs with
[] -> failwith "hd”
| x::ys -> x;;
```

Oltre il funzionale puro: Moduli

Permettono di definire ADT. La definizione ha 2 parti:

```
module Ide=(struct
    definizioni (tipi e funzioni)
end: IdeType)
```

e l'interfaccia

```
module type IdeType=
sig
identificatori (tipi e funzioni/valori) visibili all'esterno
end;;
```

Generici di Java

- In Java 1.5 viene esteso consentendo di specificare nelle definizioni delle **classi** e dei **metodi** dei **parametri** che sono istanziati su **tipi.**
- In questo modo il linguaggio ha tutte le forme di polimofismo: **overloading**, di **sottotipo**, **parametrico**.
- Interessante per le relazioni tra le vari forme di polimorfismo.
- Rimane la necessità della dichiarazione esplicita dei tipi, per la creazione degli oggetti, ma il sistema inferisce i tipi per l'invocazione dei metodi.

Polimosfismo di sottotipo e parametrico

- Le dichiarazioni di classi e metodi sono estese con parametri (formali) di tipo
- Alla dichiarazione di un parametro di tipo è possibile specificare delle condizioni che coinvolgono la gerarchia dei tipi

  <T1 extends T2> <T1 super T2>

  con T1 e T2 nomi di classi, parametri di tipo o wildcards (?)

  Es.

```
public class FList <C> { ...
public class FList <C extends Shape> {...
```

Funzioni covarianti e controvarianti

Se T è un insieme su cui è definito un ordinamento e f è una funzione da T -> T si dice che f è :

covariante se rispetta l'ordinamento, cioè

se t1<t2 => f(t1)<f(t2)

controvariante se non rispetta l'ordinamento, cioè

se t1<t2 => f(t2)<f(t1)

La proprietà si può applicare anche nel caso di ordinamento parziale (in matematica in questo caso si parla di funzioni monotone o antimonotone).

Se la funzione ha più parametri la proprietà si riferisce a ciascuno dei parametri indipendentemente.

Funzioni covarianti e controvarianti

In Java (in tutti i linguaggi con polimorfismo di sottotipo e parametrico) definizioni di classe parametriche rispetto ai tipi introducono delle funzioni sull'insieme dei tipi (polimorfismo parametrico)

La relazione di sottotipo definisce un ordinamento parziale (polimorfismo di sottotipo).

Di quali proprietà godono le funzioni sui tipi definite?

FList <Circle> <: ? FList <Shape>

dato che Circle <: Shape

Intuitivamente verrebbe da pensare che siano covarianti ma...

Tipi modificabili / non modificabili

I costruttori di tipo non possono essere covarianti se il tipo è modificabile.

Tipi non modificabili (in Java)

- predefiniti: String
- definiti dall'utente dipende dai metodi (non devono avere metodi modificatori) e l'implementazione deve essere nascosta.

(Importante per gli esercizi di trasformazione Caml $\leftrightarrow$ Java)

Costruttori di tipo NON covarianti in Java

La definizione di tipo generica NON preserva la gerarchia di sottotipo. es:

```
ShapeList <Shape> FC= new ShapeList<Shape>();
ShapeList <Circle> FC1= new ShapeList<Shape>();
```

anche se Circle <: Shape

ShapeList <Circle> $\not<:$ ShapeList <Shape>

Es. (Geometria)

Eccezione per gli array: Integer [] è sottotipo di Object [], risultato errore di tipo a runtime!

Relazione di sottotipo sui tipi funzione

(Covarianza - controvarianza)

Se

f : T1 -> T2 e g : T3-> T4

inoltre T1<:T3 e T4<: T2 allora

T3->T4 <: T1->T2

dominio controvariante

codominio covariante

In Java il problema si pone per i metodi ridefiniti che possono avere un codominio sottotipo (del metodo definito nella superclasse) e un dominio supertipo. La controvarianza eè considerata controintuitiva e percio` i domini devono essere lo stesso tipo. Es. Punto