**Emerging Programming Paradigms**

[ lezione 27 aprile ]

L’obiettivo è utilizzare le varianti polimorfe per risolvere l’expression problem, al momento queste sono implementate da OCaml e sono presenti tecniche simili in Haskell.

**Expression Problem**

Un programma archetipo (come i filosofi a cena per la concorrenza), ovvero un problema che occorre spesso in programmazione. E’ molto importante perché per molto tempo è stato utilizzato per giustificare la programmazione OO basata su classi.

Definizione: (da wikipedia) L’**expression problem** ha come obiettivo il definire un tipo di dato per casi, in cui è possibile aggiungere nuovi casi al tipo di dato o aggiungere nuove funzioni che lavorano con questo tipo, senza ricompilare il codice esistente e mantenendo la sicurezza dei tipi statici (ad esempio senza cast).

E’ un problema essenzialmente di tipaggio.

I linguaggi *funzionali* che non sono basati su classi OO e tipati, hanno storicamente molta difficoltà a risolvere (o non riescono) l’expression problem.

La soluzione con la programmazione OO utilizza 4 delle 5 caratteristiche che le classi implementano e che vengono a volte elencate come le 5 features delle classi che caratterizzano un linguaggio basato su classi:

- **Inheritance**: una sorta di cut-and-paste implicito del codice dalla classe ereditata a quella ereditante.

- **Subtype polymorphism**: una funzione è polimorfa nel senso che lavora con un certo oggetto in input appartenente ad una classe e può quindi anche funzionare su tutti quegli oggetti che sono sottoclassi, ovvero implementano gli stessi metodi con la stessa semantica, e magari possono implementare qualcosa in più.

- **Dynamic dispatch**: l’unica vera caratteristica propria della programmazione a oggetti, fa riferimento al fatto che staticamente quando si invoca un metodo non si sa che codice verrà eseguito, a runtime verrà fatto un lookup che recupera un puntatore e passa in input l’argomento opportuno.

- **Open recursion**: una ricorsione ma nell’ipotesi di mondo aperto. Ad esempio quando un oggetto in una classe ha come tipo di dato ricorsivo fatto anche dalla classe stessa (es. una lista ha una testa come coda nuovamente un oggetto lista). In realtà non è intesa solo classe identica bensì qualsiasi sua sottoclasse (intesa come inheritance e non subtyping)..

- **Encapsulation** (**non** utilizzata per implementare l’expression problem): i campi vengono mantenuti nello scope della classe ed è possibile accedervi a partire dai metodi e il dato è quindi protetto da chiamate esterne.

**Static polymorphism (dispatch)**: lo stesso nome di un metodo è **overloaded** con tipi, numero di parametri o tipo di ritorno differenti nella **stessa classe** (signature diversa). La chiamata di metodo opportuna viene risolta a tempo di compilazione.

**Dynamic polymorphism (dispatch)**: lo stesso metodo è **overridden** con la stessa signature in **classi differenti**. Il tipo di oggetto su cui il metodo verrà invocato non è conosciuto a tempo di compilazione ma sarà deciso a runtime.

Grazie alle varianti polimorfe è possibile risolvere l’expression problem senza fare uso di nessuna delle tecniche indicate sopra.

L’expression problem consiste in **due problemi** da risolvere senza usare gli elementi mostrati sopra, ma **riusando codice** esistente. Il riutilizzo del codice deve essere fatto senza cut-and-paste in quanto se c’è un bug in una porzione di codice verrebbe copiato altrove, inoltre lo stesso codice copiato e incollato può essere corretto in un punto ma non in un altro.

Ci sono due definizioni del expression problem, la prima più facile della seconda (che è un’estensione della prima). Per esempio in Haskell è possibile risolvere la prima ma non la seconda.

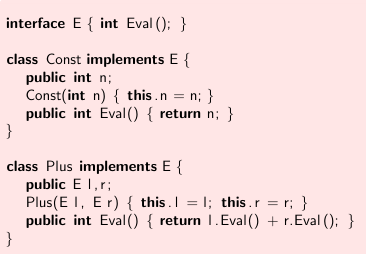
*Problema 1*: bisogna avere un tipo di dato che rappresenti la sintassi di espressioni aritmetiche (es. somme, prodotti, variabili, costanti, ...) ed avere funzioni definite su queste espressioni (es. una funzione per stampare o valutare un’espressione). Dopodichè è necessario riuscire ad aggiungere **nuove funzioni** e **nuove forme** (costruttori) delle espressioni(ad esempio se si hanno solo somme e sottrazioni si devono poter aggiungere anche le divisioni) riutilizzando codice esistente senza modifica in place e copia incolla.

*Problema 2*: dati gli stessi elementi di prima (espressioni e funzioni su di esse), è necessario che sia possibile definire l’**unione non disgiunta** tra due linguaggi.

Per esempio il primo tipo di dato contiene costanti e somme il secondo tipo di dato contiene variabili e somme, dev’essere possibile ottenere un terzo tipo di dato che rappresenti costanti, variabili e somme.

**In Java**

Si definisce un’interfaccia E che rappresenti l’interfaccia di tutte le espressioni, ogni costruttore diventa una classe che implementa l’interfaccia.



Le chiamate di Eval ricorsive (es. l.Eval e r.Eval) utilizzano *dynamic dispatch*. Aggiungere nuovi costruttori è semplice (basta aggiungere nuove classi che implementino E), difficile è aggiungere nuove funzioni. La soluzione sarebbe quindi creare una nuova interfaccia che erediti dalla precedente (E) e vi aggiunga le funzioni che si vogliono integrare, dopodichè le classi dei costruttori precedenti vanno reimplementate, ereditando da quelle presenti per evitare di reimplementare Eval() le quali implementerebbero la nuova interfaccia con l’implementazione delle nuovi funzioni.

La soluzione funziona ma è estremamente verbosa e offuscata, il codice ha un sacco di boilerplate (codice incluso in molte posizioni senza nessuna o con minime alterazioni). Se si vuole lavorare in ipotesi di mondo chiuso non si ha nessun modo per farlo (perché si sarebbe dovuto fare sealing ma poi non si poteva estendere l’interfaccia, andava reimplementata da zero ad ogni funzione da aggiungere), si è obbligatoriamente in ipotesi di mondo aperto.

**In ML**

Non è possibile risolvere il problema, in quanto l’unica soluzione possibile sarebbe l’utilizzo di cut-and-paste e non vi sarebbe riuso di codice.

Il problema è che in OCaml sul tipo di dato che ci servirebbe per risolvere l’expression problem (e.g. type e = Const of int | Plus of e \* e) vi sono due ipotesi di mondo chiuso:

- chiusura in *larghezza*: non si può aggiungere nient’altro a destra del dato (ad esempio non si può aggiungere dopo “Plus of e \* e” un caso “... | Mult of e \* e…”).

- chiusura in *profondità*: gli elementi e di “Plus of e \* e” sono limitati a loro volta, potranno essere solo Const o Plus.

- Risolvere la **chiusura in profondità**

Il problema della chiusura in *profondità* prende anche il nome di **ricorsione chiusa**. Per risolvere questo problema è sufficiente aprire la ricorsione. Una tecnica banale può essere passare al tipo e anche un parametro di tipo ‘a (alpha) e riscrivere poi la (ex)ricorsione come “Plus of ‘a \* ‘a” in questo modo il tipo di dato non è più ricorsivo, perché usa alpha che è un tipo esterno invece che il tipo che stavamo definendo (e). Dopo questa modifica anche il metodo *eval* di una espressione dovrà essere riadattato, infatti oltre a prendere un’espressione prenderà anche una funzione aggiuntiva f che verrà eseguita su a’ (dove prima c’era la ricorsione) per valutarla (trasformarla in intero). I benefici del pattern matching sono ancora presenti, ma la ricorsione non è più presente, e quindi il tipo è più generale.

NOTA: la ricorsione aperta può essere chiusa in ogni momento con l’operatore di punto fisso. Ad esempio definendo il tipo e1 che è la chiusura di questo mondo (e.g. type e1 = e1 e), e1 non è altro che il tipo e che va in ricorsione su e1. L’alpha visto prima ora è rappresentato da e1 che è a sua volta di tipo e, chiudendo così la ricorsione. Per chiudere eval è sufficiente passargli come f la funzione eval stessa (e.g. let rec eval1 x = eval eval1 x).



- Risolvere la **chiusura in larghezza**

Risolvere questo problema non è possibile tramite il solo utilizzo di dati algebrici. È possibile un’approssimazione, usata (ad esempio da Haskell) per risolvere la prima variante dell’expression problem (ovvero aggiungere nuove funzioni e nuove forme a un tipo di dato). Come prima oltre a passare un parametro alpha, viene passato anche un parametro beta:

type (‘a, ‘b) e = Cons of int | Plus of ‘a \* ‘a | More of ‘b

Se si vuole chiudere sarà sufficiente impostare ‘b come tipo vuoto, o se si vuole un prodotto si imposta ‘b con un tipo che ha un prodotto. Qui non si avrebbe però un vero e proprio Mult aggiunto ma un More of Mult.

Per risolvere in modo migliore la chiusura in larghezza sono utilizzate le **varianti polimorfe**.

**Varianti polimorfe**

L’idea è quella di decomporre un tipo di dato algebrico in due costruzioni differenti:

1. una è il **labeling** dei prodottidi tipi. Un tipo di dato algebrico è un coprodotto (quindi una somma disgiunta) di prodotti (quindi di record) e queste somme sono etichettate. Disgiunte perché quando si prende un record viene etichettato con un atomo. L’operazione è appunto l’associare ad un qualsiasi tipo di dato del linguaggio un etichetta. Operazione facile in Erlang (usi un atomo come etichetta e tutto il resto è il dato), il problema è farlo in un linguaggio tipato.

2. la seconda operazione è l’**unione non disgiunta** di dati già etichettati (essendo già etichettato il dato, non si confonde la provenienza del tipo di dato, a meno che non si usi la stessa etichetta, come avviene nella seconda variante dell’expression problem).

Cioè una variante polimorfa la puoi ottenere in questi due modi.

In breve le varianti polimorfe sono simili alle varianti normali con l’unica differenza che i costruttori non sono più legati ai tipi, ma esistono anche indipendentemente. Esempio:

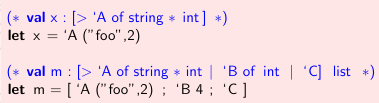
Variante normale **type** e = Cons of int | Plus of e \* e;

Variante polimorfa **type** e1 = [`Cons of int | `Plus of e \* e];

Il costrutto Plus (3, 5) lo puoi usare in giro solo dopo averlo dichiarato all’interno di e, mentre ‘Plus (3, 5) lo puoi usare anche prima di averlo dichiarato in e1 (è indipendente).

Una volta che si hanno queste due operazioni è possibile codificare i tipi di dati algebrici però senza *uguaglianza* strutturale, bensì *nominale* (grazie all’etichetta).

Esempio in OCaml:



Come etichette qui si utilizzano atomi che iniziano con la lettera maiuscola. Qui gli apici sono diversi dagli apostrofi usati per alpha e beta.

In let x = ‘A(“foo”, 2) non viene in precedenza definito cosa sia ‘A quindi è OCaml che prova ad inferire un tipo. Precisamente per le varianti polimorfe ha un modo più articolato per descrivere il tipo, qui inferisce che ‘A è un tipo che ha una stringa e un intero, con il > indica che x ha ALMENO la forma (stringa, intero), ma non esclude che possa avere anche altre forme (che x oltre ad ‘A possa avere anche ‘B o altro). Attenzione che ‘A è il label.

In let m = [ ‘A (“foo”, 2) ; ‘B 4 ; ‘C ] (liste omogenee, tutti gli elementi stesso tipo) il tipo che OCaml inferisce è che m contiene almeno (>) il tipo ‘A con una stringa e un intero oppure un ‘B con un intero, oppure un ‘C (essendo le liste omogenee il compilatore immagina che ogni elemento della lista possa avere ALMENO uno dei tipi inferiti).

Negli ADT l’utente descrive tutte le possibili forme e quando si scrive il codice il compilatore verifica l'esaustività e l’assenza di duplicati. Con le Varianti Poliforme il compilatore deve capire attraverso la type inference quali sono le forme utilizzabili (duck typing).



La “predizione” scritta in blu può essere ‘letta’ nei seguenti modi:

1. Ha la forma ∀α.[ ‘A of string \* int | α ], quindi ‘A che contiene string\*int oppure α. Alpha quantifica universalmente (polimorfismo uniforme) su tutte le possibili sequenze di coppie (nome, tipo). Alpha potrebbe essere istanziata come ‘B of int o ‘C.

Il poliformismo uniforme visto finora prevedeva che i “per ogni” quantificassero solo su tipi (es. per ogni tipo alpha, per ogni tipo beta, ecc) ora invece non si quantifica su un tipo alpha ma su una sequenza di coppie nome-tipo (chiamate row). Questa forma di poliformismo di chiama appunto **polimorfismo di riga** (polimorfismo uniforme in cui invece dei tipi si usano le coppie tag-tipo, chiamate righe).

Si può vedere il tag come un’informazione aggiuntiva sul tipo, così puoi avere due tipi con la stessa struttura, tipo una coppia di stringhe, ma con tag diversi, quindi il **confronto** tra due istanze di questi due tipi rimane **strutturale** (ovvero sono considerati uguali), ma quando devi ad esempio creare un nuovo tipo che li accomuni entrambi se hanno tag diverso sono considerati tipi distinti, se hanno tag uguale verrà messa solo una loro istanza nel tipo risultante (questa è l'**unione** **non disgiunta**).

Ma a cosa serve il tag?  
type rigaA: 'NomeCognome : (string, string) | 'CittaProvincia : (string, string)

type rigaB: 'NomeCognome : (string, string) | 'NatoLuogo : (string, string)

se vuoi creare il tipo rigaC unendo rigaA e rigaB, essendo varianti polimorfe si usa l'unione non disgiunta:

type rigaC: 'NomeCognome : (string, string) | 'CittaProvincia : (string, string) | 'NatoLuogo : (string, string)

Quindi i due 'NomeCognome avendo lo stesso tag vengono considerati uguali e non vengono ripetuti, gli altri hanno tag diversi e pur avendo lo stesso tipo (una coppia di stringhe) vengono aggiunti lo stesso distintamente. Però se vai a fare dei confronti sul tipo di 'CittaProvincia e 'NatoLuogo ti dirà che sono uguali (il confronto è strutturale), questo è utile tipo per il casting, cioè eseguire una funzione che prende una coppia di stringhe su 'CittaProvincia o 'NatoLuogo andrà bene comunque

Le varianti polimorfe sono un’applicazione del polimorfismo di riga.

2. un’altra lettura possibile di “let x: ‘A(“foo”,2)” (leggermente più corretta) è che ha come forma ∀α.α ovvero per ogni riga α ha come tipo α. È soggetto a due vincoli:

- che α sia una riga (ovvero un’unione di tipi taggata).

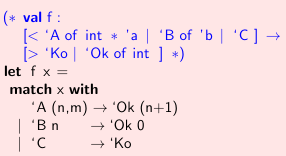
- che in questa lista di tipi α può essere ci sia almeno [‘A of string \* int].

Questo è chiamato **polimorfismo bounded**, si quantifica su tutto a patto che soddisfi dei vincoli (chiamati bound).

3. un’ultima lettura possibile (non corretta) è che ha la forma ∀α.α dove α è un sottotipo del tipo che ha solamente [ ‘A of string \* int ] (mancano ‘B … e ‘C, ecc). Questo è errato perché il tipaggio implicito violerebbe il tipaggio principale (il compilatore inferisce il tipaggio sbagliato).

[Blog post interessante per capire un po' le polymorphic variant](https://2ality.com/2018/01/polymorphic-variants-reasonml.html)

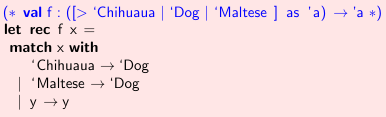
Esempio 1:



Il compilatore deve capire che il dominio della funzione può solo essere un sottoinsieme di ‘A, ‘B, ‘C è impossibile passargli ‘D (in quanto non contemplato nel match).

L’output inferito è [> ‘Ko | ‘Ok of int] indica che come output saranno presenti ALMENO questi due tipi. L’input inferito è [< ‘A of int \* ‘a | ‘B of ‘b | ‘C], il simbolo minore indica che AL MASSIMO si potranno avere questi 3 tipi, escludendo la presenza di un ipotetico ‘D (questo è coerente con le opzioni contemplate dal match).

Esempio 2:

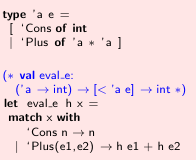
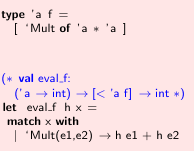


In questo caso il tipo inferito di input è ALMENO ‘Chihuahua, ‘Dog o ‘Maltese (ma può avere anche altro dato che è presente quel y → y). L’output deve contenere ALMENO ‘Dog ma può avere anche qualsiasi altra cosa. Per rappresentare questo concetto viene chiamato l’input come ‘a (una sorta di semplice alias) e viene dato in output sempre questo ‘a (perché di fatto sono i valori che possono uscire in output).

La realtà presenta un problema, quando si scrive un programma scorretto si riceve un errore di tipo. L’errore di tipo normalmente è “la funzione si aspetta questo tipo di input ma gli viene passato quest’altro”. Se non vengono specificate le forme ma si usano in maniera massiccia le varianti poliforme, si riceve come errore un input atteso lunghissimo e illeggibile (tipo “la funzione si aspetta questo, oppure quello ma fatto così oppure cosà, … e invece è stato passato {altra descrizione illeggibile}” ). Questo crea grossa confusione nel capire cosa realmente è andato storto. Spesso è necessario restringere a mano qualche tipo.

Tornando all’expression problem, abbiamo visto come risolvere la *chiusura in profondità*, mentre per risolvere la *chiusura in larghezza* si utilizzano appunto le **varianti polimorfe**.

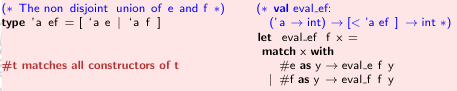
Ovvero dato un tipo **e** si vuole estendere con il tipo **f** (ovvero si vuole un tipo che contenga sia Cons che Plus ma anche Mult). Il tipo di e ed f viene definito con la dicitura delle varianti polimorfe (potrebbe anche essere inferito è scritto esplicitamente solo per chiarezza).

eval\_e ed eval\_f sono scritte come viste in precedenza, solo che il loro tipo stavolta sarà:

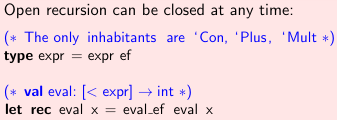
eval\_e: prende in input un qualunque tipo che ha AL MASSIMO un alpha espressione (quindi che ha al più Cons e Plus).

eval\_f: prende in input un tipo che ha AL MASSIMO Mult.



L’unione dei due tipi per risolvere l’expression problem avviene tramite unione non disgiunta di e ed f. Quindi si crea un nuovo tipo ef in cui vengono elencati tutti e tre assieme (Cons, Plus e Mult), e di alpha è già una riga, f di alpha è già una riga, mettendole in or (cioè o uno o l’altro) si effettua l’unione. Si definisce anche una nuova eval\_ef il cui tipo sarà prendere in input AL MASSIMO un tipo ef (ef è un unione di Cons, Plus e Mult) e restituisca un intero.

#e indica che matcha tutti i costruttori del tipo e (ovvero Cons e Plus) quando questo match avviene si chiama la eval\_e che è in grado di processare questi tipi. In questo modo si è riutilizzato codice.

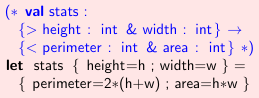
Qui, a differenza della soluzione con classi, è possibile chiudere il mondo all’occorrenza:

In modo simile a prima si definisce un espressione che ha il tipo dell’espressione stessa, saranno accettati tipi solo di quel genere.

NOTA: non è utilizzato in alcun modo dynamic dispatch a runtime, perchè il compilatore sa tutte le funzioni che saranno invocate e può quindi ottimizzare il codice.

Viene usato poliformismo basato su istanziazione di quantificatori universali, NON su sottoclassi.

Il **duale** delle varianti polimorfe (unioni non disgiunte di valori taggati con tipaggio strutturale e polimorfismo generico e type inference) saranno quindi non unioni disgiunte, bensì prodotti cartesiani non disgiunti, il costrutto ottenuto è chiamato **extensible record**. Questi sono prodotti (record) di valori taggati (campi) se due record hanno campi in comune questi vengono identificati. Ad esempio:



Il tipo prende in input un record e in output un record (identificati dalle graffe). Quello in input ha un certo numero di campi ma deve avere ALMENO un altezza, che è un intero e una larghezza che è sempre un intero. In output un record che ha AL MASSIMO un perimetro e un’area.