**Emerging Programming Paradigms**

[ lezione 04 maggio ]

L’inheritance è un costrutto nativamente bacato, un buon sostituto è il *polimorfismo parametrico* (generics e templates), un costrutto emergente è invece il *polimorfismo parametrico di riga*, che permette di abbandonare completamente il concetto di inheritance.

Il **polimorfismo di riga** consiste nell’avere *polimorfismo parametrico* (templates in C++, generics in Java, a livello logico è visto come un “per ogni”) ma anche *variabili di riga*.

Una **variabile di riga** è una variabile che può essere istanziata con una mappa (un insieme di coppie) etichetta-tipo.

Esempi di utilizzo del polimorfismo di riga sono le varianti polimorfe in OCaml, i record estensibili in Elm e gli oggetti funzionali in OCaml.

**Extensible Records**

Una **variante polimorfa** è un’unione non disgiunta (coprodotto) di valori taggati, tipata con polimorfismo di riga (ogni etichetta nel tipo rappresenta una possibile alternativa nel dato).

Un **record estensibile** è un record (prodotto cartesiano non disgiunto) tipato con polimorfismo di riga (ogni etichetta rappresenta un campo e sono tutti a disposizione contemporaneamente, mentre nella somma disgiunta si ha una sola possibilità in input).

I prodotti e coprodotti sono costrutti duali, derivanti dal mondo della logica. Gli extensible record sono stati introdotti nel 1987 da Wand e poi abbandonati a favore degli oggetti. Oggi sono alla base di Elm e integrabili come libreria in Haskell.

Sono una versione fortemente tipata (quindi con grosse garanzie) di quello che avviene nella stragrande maggioranza di linguaggi non tipati (Javascript, Python) in cui il tipo fondamentale è la mappa. Ad esempio in Javascript è possibile definire un record di elementi nome:valore ed è possibile chiamare una funzione su tutti quei record (chiamati oggetti in javascript) che espongono *almeno* i campi richiesti dalla funzione (duck typing). Il modo più corretto per rappresentare questi tipi sarebbe tramite polimorfismo di riga.

**Extensible records in pseudo-sintassi**

let f r = r.nome ^ “ “ ^ r.cognome

(^ concatenazione) se si prova a tipare questa funzione l’output sarà una stringa, mentre l’input sarà un record tipato con una sequenza di campi che contengono almeno un campo nome e un campo cognome: val f : {> nome : string ; cognome : string} → string questo è zucchero sintattico per {nome : string ; cognome : string ; ρ} con ρ una row variable.

Un record estensibile supporta tre tipi di operazioni:

- **Accesso** a un campo: val.name : {name : ‘a ; ρ} → ‘a

- **Aggiornamento** di un campo: val.set\_name : {name : ‘a ; ρ} as ‘b → ‘a → ‘b ritorna un nuovo record con gli stessi campi ma in cui uno è stato modificato.

- **Aggiunta** di un campo: val.extend\_name : {ρ} as ‘b → ‘a → {name : ‘a; ρ} l’output è un record con gli stessi campi di quello passato in input con in più il campo specificato.

Zucchero sintattico: dato r un extensible record di tipo {ρ}, {r with name = “Claudio”}

Il vantaggio principale dell’utilizzo di record estensibili è lo **static dispatch**, ovvero a tempo di compilazione il compilatore sa quale funzione verrà chiamata, questo gli permette di ottimizzare pesantemente il codice. La correttezza del codice quindi non dipende da come questo verrà usato ma da come è stato scritto. Possiamo quindi parlare di ipotesi di mondo chiuso.

* **no side effects**: questo costrutto permette di avere oggetti funzionali, ovveri i campi sono immutabili e ogni modifica ritorna un nuovo oggetto che condivide i campi non toccati in memoria.
* **polimorfismo**: non c’è bisogno di inheritance, classi, interfacce, ecc. con l’ereditarietà la classe figlia può modificare i campi del padre con l’overloading, qui non è possibile, si sa esattamente cosa può modificare cosa.
* **static dispatch**: a tempo di compilazione il compilatore sa quale codice chiamare.
* **no data hiding**: nel caso sia necessario si possono usare altri meccanismi (chiusure, esistenziali, …)

Un oggetto, a differenza di un record estensibile, fornisce **dynamic dispatch**.

Un **pre-metodo** è una funzione che prende in input un oggetto (un record) e opera su di esso, come nome dell’input spesso si usa **self** o **this** per far riferimento all’oggetto stesso.

Di norma i pre-metodi sono campi dell’oggetto (record) stesso, quindi dall’oggetto estraggo il codice con il punto e poi passo in input l’oggetto (dynamic dispatch).

Pertanto per invocare il pre-metodo m dell’oggetto o si usa (o.m o) e il codice invocato non è noto a compile time.

Il dynamic dispatch è più lento, permette meno ottimizzazioni ed è in ipotesi di mondo aperto quindi non si ha alcuna garanzia sugli invarianti e di conseguenza meno sicurezza.

**Oggetti in OCaml**

OCaml non implementa record estensibili, bensì oggetti. Questi oggetti sono in qualche modo diversi dai record estensibili.

Gli oggetti in OCaml hanno *campi* e *metodi*, i campi sono visibili solo dai metodi (**data hiding**), quindi se si vuole dynamic dispatch si ha “per forza” anche data hiding.

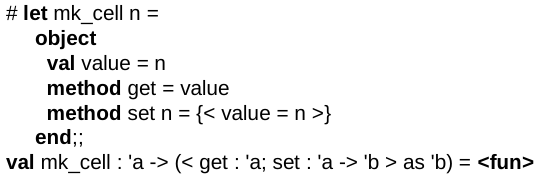
I metodi non sono mutabili, ma lo sono solo i campi, questo permette un’ottimizzazione in quanto

se più oggetti usano metodi uguali è possibile farli puntare ad un’unica tabella comune a tutti.

Questi oggetti hanno in comune con i record estensibili che il polimorfismo di riga viene usato per i tipi degli oggetti. Si parla di **oggetti funzionali** quando nessun campo è mutabile. Inoltre gli oggetti non sono estensibili, non è possibile aggiungere campi o metodi dopo la creazione dell’oggetto. Infine il dispatch dei metodi è sempre dinamico, a differenza dei record estensibili.

NOTA: questo è un problema limitato perché se si vuole dynamic dispatch si invoca un metodo di un oggetto, se si vuole static dispatch si scrive una funzione che prende in input l’oggetto.

Esempio in OCaml:

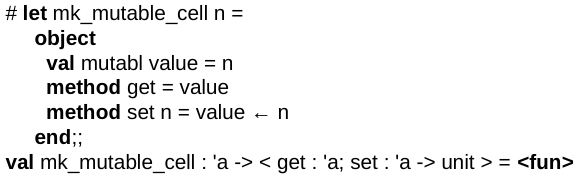


Set restituisce beta in quanto è stato indicato come tipo dell’oggetto (as ‘b). Questo tipo ricorsivo è importante perché il metodo set può così lavorare su oggetti con più campi, oltre ai due get e set indicati, e funzionerebbe comunque bene.

NOTA: il tipo dei valori non è esposto nell’interfaccia dell’oggetto (data hiding).

**Oggetti mutabili**

È possibile specificare che un campo è mutabile (in place) grazie alla keyword **mutable**.



La set restituisce un unit (un tipo con solo costruttore), ovvero non dà niente in output perché la mutabilità avviene in-place.

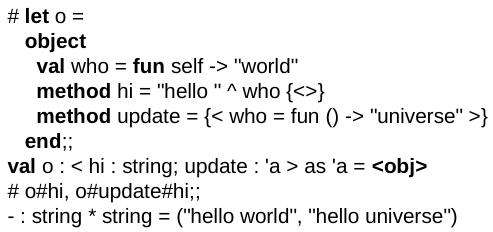
La **mutabilità** ha sempre cattive conseguenze sulla correttezza:

- gli invarianti (condizioni che devono essere verificate ovunque nel codice) possono essere facilmente rotti (anche se il data-hiding aiuta a conservarli)

- le funzioni hanno side-effects (anche se localizzati ai campi dei metodi)

- per testare la correttezza di un programma è necessario non solo passargli l’input giusto ma anche l’ambiente giusto e verificarne i side-effects sull’input e sull’ambiente.

NOTA: i metodi non sono modificabili ma i pre-metodi usati come campi si:



Ad esempio qui si cambia il comportamento di un metodo (hi) codificandolo come pre-metodo.

Il consiglio è: usare i metodi quando si ha bisogno di late-binding, altrimenti funzioni.

**Subtyping**

Grazie al polimorfismo di riga difficilmente si ha bisogno di subtyping in OCaml, proprio perché se si vuole utilizzare qualcosa di più “specifico” è sempre possibile, purché abbia i “requisiti minimi” (i campi utilizzati dalle funzioni che usano quell’elemento).

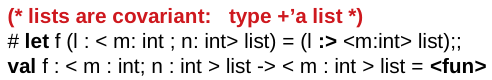
Tuttavia a volte è necessario, ad esempio se si vuole creare una lista con due oggetti, uno che ha i campi m ed n, mentre l’altro che ha solo il campo m, la lista richiede che tutti gli elementi abbiano lo stesso tipo. Per mettere entrambi gli oggetti nella stessa lista bisogna “dimenticare” l’esistenza del campo n nel primo oggetto.

Esiste un operatore di subtyping esplicito chiamato **downcast** con la keyword **:>**. A runtime è sempre corretto, perché si toglie, non si aggiunge. Inoltre la restrizione di tipo è sempre irreversibile, non è possibile poi effettuare upcast (ovvero riaggiungere il campo precedentemente tolto), altrimenti a runtime si romperebbe (dynamic casting).

In presenta di sottotipaggio (downcast) è necessario introdurre la nozione di **covarianza**.

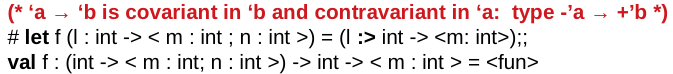
Un tipo è *covariante* (si indica con un +) quando è **monotono** come costruttore di tipo.

*Monotono* significa che se alpha1 è un sottotipo di alpha2 (a1 <: a2) allora un alpha1 list è un sottotipo di un alpha2 list, quindi la funzione list è monotona.



I vari costruttori di tipi hanno varianze differenti, a seconda di come si implementa il tipo di dato la varianza può essere *covariante*, *controvariante* o nessuna delle due (invariante).

Ad esempio ‘a → ‘b è covariante in ‘b ma controvariante in ‘a, cioè è monotona sull’output ma antimonotona sull’input, sintatticamente va da meno alpha a più beta: -’a → +’b.



L’input è un intero e l’output m ed n, può essere ristretta vedendola come una funzione con in input un intero e in output solo m. Sull’input è il contrario, non si può prendere una funzione che prende in input m ed n e vederla come una funzione che prende in input solo m. In input la funzione è controvariante.

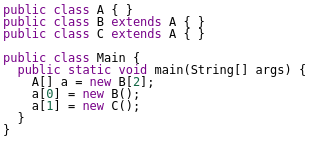


Quando si dichiara un tipo di dato il fatto che sia covariante o controvariante non è una scelta del programmatore bensì dipende da quali operazioni vi sono definite sopra.

**Problemi con Java**

Java ha costruttori di tipo (array, liste, matrici, code, …) la differenza tra questi e quelli di OCaml è la mutabilità, quelli di java non solo permettono di leggere gli elementi al suo interno ma anche di (sovra)scriverli questo rende le liste/array non più covarianti o controvarianti.

Un esempio è che in Java si può creare un array T[] che conterrà elementi con tipo T o sottotipi di T. Tuttavia Java afferma che un array di tipo S[] è un sottotipo dell’array T[] se S è sottotipo di T. Quindi in Java se si crea un array di tipo S[], lo si assegna a T[] (è possibile perché S[] è considerato sottotipo di T[]) poi si modifica un elemento dell’array risultante con un altro sottotipo di T che non è anche sottotipo di S il compilatore lo accetta, ma a runtime viene sollevata un’eccezione.



[viene dato errore a runtime ma non a compile-time]

Java non nasce con polimorfismo uniforme, ma solo con inheritance, i generics sono stati aggiunti dopo. Java considera i tipi che fanno uso dei generics come invarianti, anche se dovrebbero essere covarianti e controvarianti. Ad esempio:

List<Integer> ints = new ArrayList<Integer>();

List<Number> nums = ints;

da errore a tempo di compilazione, anche se non dovrebbe in quanto Integer è sottoclasse di Number.

Per avere polimorfismo di riga in Java bisogna indicare “per ogni tipo T che eredita da una certa classe”, a differenza di OCaml in cui si dice “per ogni tipo T che ha certi metodi”.

Utilizzando l’espressione ? extends T si indica una riga, qualsiasi classe che abbia tutti i metodi che vuole, purchè abbia *almeno* quelli presenti in T. In questo modo il polimorfismo di riga funziona in quanto si è aggiunta **covarianza**, però non è più possibile (in Java) *scrivere* sull’elemento che utilizza polimorfismo di riga.

Utilizzando l’espressione ? super T si indica una riga, qualsiasi classe che abbia *al massimo* i metodi presenti in T. In questo modo si aggiunge **controvarianza**. In questo modo il polimorfismo di riga funziona ma non è più possibile (in Java) *leggere* l’elemento, solo scriverlo.

Covarianza e controvarianza in input/output non sono standard, dipendono dal costrutto in questione. Ad esempio in OCaml le liste sono *covarianti* in input (una lista di Gatto è un sottotipo della lista di Animale, in quando la lista Gatto preserva tutti gli elementi presenti nella lista Animale), le funzioni sono *controvarianti* in input (una funzione Animale -> String è sottotipo della funzione Gatto -> String perché la prima potrà sempre lavorare anche con Gatto ma il contrario non sarebbe possibile)

Ciò che è stato visto finora rendono OCaml un linguaggio **object based**, ovvero sono presenti gli oggetti, è presente una qualche forma di polimorfismo, sia esso di riga o sottotipaggio, è presente dynamic dispatch ma sono assenti classi e inheritance (con i loro relativi problemi). Javascript come OCaml ha un nucleo object-based.

I linguaggi derivati da Simula (C++, Objective C, Python, Java, C#, …) sono **class based** (anche chiamati object oriented).

Origine degli oggetti:

- Subito si è pensato agli oggetti come a piccoli programmi distinti che risolvessero ognuno uno specifico problema, e tramite uno scambio di messaggi (i metodi erano chiamati messaggi) comunicassero tra loro.

- Successivamente si è voluta portare tutta la logica che stava dietro all’ingegneria del software fino al livello di programmazione, creando così la struttura a classi e tutto il filone relativo.

I linguaggi recenti (Go, Rust, Julia, …) sembrano aver abbandonato l’approccio su classi (inheritance, override, …) per far spazio a traits, mix-in, e in generale nuovi costrutti più logicamente corretti.

OCaml ha anche costrutti di natura class-based.

**Classi in OCaml**

Una classe è un meccanismo per il riuso del codice tramite ereditarietà. La dichiarazione di una classe non è un oggetto del prim’ordine (non può essere presa in input, data in output, ecc). Esempio di creazione di una classe:

class ['a] stack vs =

object

val l = vs

method push (x : 'a) = {< l = x::l >} end;;

La classe viene tipata come:

class ['a] stack :

'a list -> object val l : 'a list method push : 'a -> 'b end as ‘b

L’operatore new prende una classe e restituisce un oggetto:

new stack;;

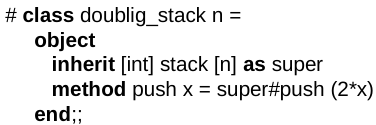
Tipato come:

'a list -> 'a stack = <fun>

In cui ‘a stack è il tipo riga dell’oggetto corrispondente allo stack. Viene restituito un oggetto contenente solo il metodo push e non anche il campo l.

**Inheritance**

Le classi hanno senso solo se vi è anche un meccanismo di inheritance. In OCaml è possibile tramite l’operatore **inherit**. Esempio:



Tipato come:



NOTA: l’unica semantica logicamente corretta per l’ereditarietà è il cut-and-paste (niente subtyping o altro).

Uno dei tanti problemi che l’inheritance crea è che *l’ereditarietà NON è subtyping*.

In nessun linguaggio di programmazione l’ereditarietà dovrebbe indurre sottotipaggio, perché a livello logico non funziona. Se un linguaggio ha questa caratteristica allora:

- o trucca il sistema di tipi per forzare le classi ereditate a essere sotto-classi, rendendo inconsistente il sistema di tipi (vengono lanciate eccezioni a run-time)

- o quando si eredita si cambiano solo alcune occorrenze del tipo della classe con quelle del sottotipo e non altre (Java).

Esempio:

Data la classe point:

class point = object(self : 'self) method eq (x: 'self) = self end;;

eq è un metodo che prende in input un oggetto x con lo stesso tipo dell’oggetto e ritorna se stesso (non usa il parametro x). Il tipo di questa classe è il seguente:

class point : object ('a) method eq : 'a -> 'a end

Data la classe colored\_point che eredita da point:

class colored\_point = object inherit point method c = 4 end;;

La classe eredita le stesse cose di point e aggiunge il metodo c (color) che restituisce sempre lo stesso valore. Il tipo di questa classe è il seguente:

class colored\_point : object ('a) method c : int method eq : 'a -> 'a end

Di fatto avviene un cut-and-paste di eq oltre che a c.

Supponiamo di scrivere la seguente funzione:

let f (x : point) = x;;

Prende in input un punto e lo ritorna. Verrà tipata come:

val f : point -> point = <fun>

A questo punto eseguire:

f(new colored\_point)

ritorna un errore perché OCaml tenta un cast da f : point -> point ad f : colored\_point -> colored\_point, tuttavia in quanto colored\_point (sebbene erediti) non è un point.

Il cast non è possibile perché una funzione colored\_point -> colored\_point non è un sottotipo di point -> point perché colored\_point è un sottotipo di point in output, e va bene perché in output è covariante, ma in input le funzioni richiedono ai tipi di essere controvarianti, quindi dovrebbe essere point un sottotipo di colored\_point, ma ciò non è vero. Infatti se accettassi colored\_point per f, in output si può ignorare il colore (per downcasting) e restituire comunque il point, ma se passassi in input ad f un oggetto che non ha un colore, crash a runtime.

Data eq : colored\_point -> colored\_point non è possibile castarla a eq’ = (eq : point -> point) perché potrebbe certo castare in output colored\_point a point rimuovendo la variabile color, ma se eq facesse uso della variabile color all’interno del suo corpo crasherebbe perché ha ricevuto in input un point che non ha il campo color.

Se invece si volesse passare da f : point -> colored\_point a g = (f : colored\_point -> point) non ci sarebbero problemi perché in input viene passato qualcosa che non ha meno campi di point (al massimo ne ha di più) e in output viene restituito qualcosa che ne ha meno, ma non è un problema basta rimuovere quelli in eccesso.