**Emerging Programming Paradigms**

[ lezione 29 aprile ]

**Recap**

Una **riga** è una sequenza di coppie che associa un *tipo* a un *nome* (rappresentato da un atomo), es. (A, int), (B, float), (C, f int -> int) è una riga. Una riga può essere vista come una mappa, ma a differenza di quest’ultima, che restituisce in output sempre lo stesso tipo, può restituire tipi diversi.

Esempio: type riga = 'Tag1 int | 'Tag2 string | 'Tag3 float  
La riga può avere tre modalità: int, string o float (ognuno col suo tag)

Esempi di costrutti che usano le righe sono gli ADT, i Record, le classi (ha un insieme di campi nome-tipo e metodi nome-tipo) e le union del C.

Nei tipi normali ci sono i tipi e i costruttori di tipi (interi, booleani, coppie, frecce, …), il poliformismo uniforme introduce delle variabili di tipo e un “per ogni” (es. ∀α α → α).

Nel **polimorfismo di riga** si introducono delle variabili che stanno per una riga (sequenza nomi-tipi) e in maniera uniforme si quantificano con un “per ogni” (es. per ogni riga succede qualcosa). Detto brutalmente, questo tipo di polimorfismo permette che un metodo utilizzi tutti i tipi che abbiano almeno le caratteristiche utilizzate nel metodo, se ne hanno di più ok no problem, se ne hanno meno non tipa.

Le **varianti polimorfe** altro non sono che un costrutto che utilizza le righe dandogli una semantica di coprodotto (unione disgiunta). Attraverso le righe si specifica quali costruttori una funzione può prendere in input o dare in output.

L**’unione disgiunta** (o somma disgiunta) di due insiemi corrisponde all'unione insiemistica, realizzata in modo da considerare distinti elementi appartenenti ad insiemi distinti. L’unione **non** disgiunta è evidente quando si ha ad esempio un tipo [‘A of int | ‘C] che viene unito ad un tipo [‘B of bool | ‘A of int] produce un tipo con [‘A of int | ‘B of bool | ‘C], quindi il tipo ‘A of int viene unito e non ripetuto. L’unione disgiunta avviene solo quando si assegnano etichette ai tipi in una riga, infatti due tipi identici ma con etichette differenti sono considerati differenti. Infatti non è possibile avere ad esempio [‘A of int | ‘A of bool] perché non si avrebbe più unione disgiunta e non sarebbe possibile fare pattern matching su quel dato.

**Tipi dipendenti**

L’idea è che un tipo possa dipendere da un termine.

Ad esempio un tipo “list n” è un tipo lista con lunghezza n, questo n può variare (ad esempio list 3 sono tutte le liste di lunghezza 3, e così via).

Un altro esempio è un tipo coppia in cui uno dei due elementi dipende dall’altro (es. (country, City[country]) ).

Con questa caratteristica è possibile avere **codice corretto**.

Aggiungendo nel sistema di tipi anche i termini è possibile scrivere tipi che rappresentino proprietà arbitrarie del codice, è quindi possibile scrivere funzioni che contengono la loro specifica, in maniera dettagliata. Ad esempio:

l1:list → l2:list \* is\_permutation l1 l2

In questo tipo viene presa in input una lista l1 e viene restituita in output una coppia in cui il primo elemento è una lista l2 e il secondo è un **elemento abitabile**, ovvero è possibile costruire un elemento di questo tipo di dato solamente se l1 è una permutazione di l2 (isomorfismo di Curry-Howard). (is\_permutation è un predicato) Quindi un dato che abbia questo tipo fornisce in output non solo la permutazione ma anche la dimostrazione che l1 sia una permutazione di l2. È chiaramente compito del programmatore assicurarsi che is\_permutation verifichi correttamente la permutazione, ma fatto questo si avrà la certezza che il tipo di dato risultate porti con sé anche la dimostrazione di correttezza.

(Da Wikipedia) Due esempi comuni di tipi dipendenti (visti anche sopra) sono *funzioni dipendenti* e *coppie dipendenti*, il primo caso ad esempio può essere un array la cui lunghezza dipende da un parametro n passato al tipo, il secondo caso ad esempio può essere una coppia (array, lunghezza) in cui la lunghezza viene passata sempre al tipo e restituita in coppia all’array.

Definizione **tipo abitabile**: un tipo è definito abitabile se ha almeno un valore. Un “*abitante*” di un tipo è semplicemente un valore con quel tipo (es. fun f x -> x+1 è un abitante del tipo int -> int)

Usando i tipi in questo modo vengono passate alle funzioni in input anche delle prove, quindi il compilatore diventa un verificatore automatico di problemi. Delegare il compito del compilatore a trovare le prove è un problema indecidibile, quindi si può solo limitare a fidarsi delle prove che il programmatore fornisce insieme al dato.

I tipi dipendenti sono stati implementati all’interno di dimostratori interattivi di teoremi e in alcuni linguaggi di programmazione.

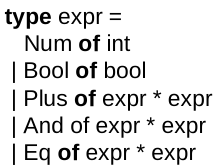
È molto difficile che i tipi dipendenti prendano piede, proprio perché è necessario scrivere più codice (quello relativo alle varie prove) e scriverlo che sia comprensibile dal compilatore che si usa, infatti è possibile scrivere codice corretto ma che il compilatore non riconosce come tale.

I Tipi di Dati Algebrici Generalizzati (GADT) sono una “versione soft” per i tipi di dati dipendenti.

**Generalized Algebraic Data Types**

Proposti nel 1994 e implementati in Haskell nel 2003, nel 2006 diventano un costrutto mainstream (sempre di Haskell). Nel 2012 anche OCaml adotta i GADT.

Prendiamo come esempio il seguente tipo di dato:



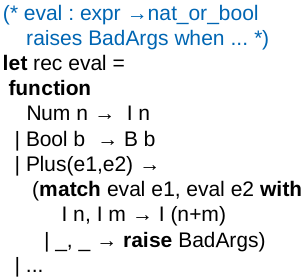
Può essere scritto come ADT, tuttavia è troppo permissivo. Cioè si riesce a rappresentare e tipare delle configurazioni di questo tipo di dato che non dovrebbero mai presentarsi a runtime, come:

let so\_bad = Plus (Bool true) (Eq (Num 3) (Bool false))

L’espressione è sintatticamente corretta ma non è ben tipata (eq tra bool e num non possibile).

Questo tipo di dato non è sufficientemente preciso.

Prendiamo allora la seguente funzione:



Qui è possibile restituire in output sia interi che booleani, in presenza di Plus viene fatta la valutazione ricorsiva di entrambi gli elementi, se sono entrambi interi, si restituisce la loro somma come intero, altrimenti si lancia un errore. Questo va riprodotto anche per And ed Eq. Il codice è brutto perché ha molti match e lancia eccezioni, inoltre l’output non si sa se sarà intero o booleano quindi chi userà il tipo dovrà fare pattern matching.

La sintassi dell GADT è la seguente:



È molto simile agli ADT ma dopo il nome del costruttore ci sono due punti e viene tipato anche il costruttore. Qui non vengono definite le espressioni ma le *alpha-espressioni*. Questo alpha potrà essere int o bool. Una bool espressione è un’espressione che rappresenta ad esempio un booleano o un intero. Nei casi di somma ed and vengono specificati non solo i tipi ma anche che devono essere uguali. Mentre in eq non importa quali siano i tipi purché siano uguali tra loro. Tutti restituiscono in output un’altra espressione che sarà anch’essa bool o int.

Ogni costruttore di un GADT può istanziare il parametro alfa in modo diverso.

*I GADT permettono di specificare esplicitamente i tipi per i costruttori di dati quando vengono definiti.*

Qui invece che avere un “per ogni” all’esterno della dichiarazione di tipo lo si ha all’interno, quindi (∀α α expr \* α expr) → bool expr. *Un ∀ a sinistra di una freccia è un ∃*. Si aggiunge quindi oltre al per ogni sui tipi anche l’esiste sui tipi. Con i GADT gli esempi mal tipati visti con ADT vengono rigettati.

*“Un ∀ a sinistra di una freccia è un ∃”* spiegazione:

**(∀α . α expr → bool) → int expr**

Qui si ha un ∀ che sta a sinistra di una freccia (quella più esterna).

In logica classica: **A → B ≡ ¬A ∨ B**

(A solo se (implica) B è lo stesso di non A or B) questo perché non è possibile avere A senza B, quindi ¬(A ∧ (¬B)) che diventa ¬A ∨ B. È ovviamente vero anche che: **¬A ∨ B ≡ A → B**

**≡ ¬∀α.(α expr → bool) ∨ int expr**

Usando la riduzione sopra, a sinistra dell’or c’è un non per ogni.

In logica classica: **¬∀x.P(x) ≡ ∃x.¬P(x)**

Ovvero se non è per ogni x significa che esisterà almeno un x per cui ciò non è valido (e viceversa).

**≡ ∃α.(¬(α expr → bool) ∨ int expr)**

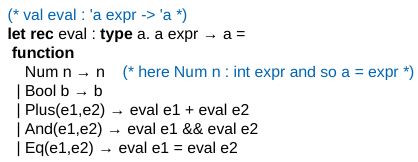
Si è messo l’or (che prima era esterno al per ogni) all’interno dell’esiste, perché c’è una legge in logica classica che dice: **(∃α.P) ∨ Q ≡ ∃α (P ∨ Q)**

**≡ ∃α.((α expr → bool) → int expr)**

Applicando la legge in logica vista per prima (**¬A ∨ B ≡ A → B**)

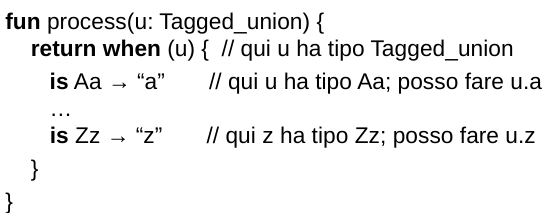
Quindi un per ogni che sta a sinistra della freccia, diventa un esiste quando viene portato fuori dalla freccia.

Inoltre la eval con i GADT diventa:



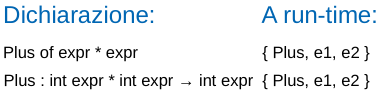
I GADT rendono la type inference indecidibile quindi è necessario indicare manualmente il tipo, in questo caso viene indicato type a. a expr → a (letto come “per ogni tipo a prendi un alpha espressione e restituisci un valore di tipo alpha, esempio data un’espressione con intero restituisci un intero).

Kotlin fa una cosa simile, nei rami del case il tipo delle variabili mutava a seconda di cosa era stato visto con le guardie del case.



Il compilatore staticamente analizza il codice e cambia il tipo delle variabili.

Un GADT è isomorfo a runtime all’ADT ottenuto cancellando i parametri istanziati dai costruttori.



A runtime GADT e ADT hanno la stessa forma.

Si possono usare questi vincoli di tipaggio per imporre invarianti sui dati, ad esempio costruire un albero red and black in modo che a tempo di compilazione lancia un errore se le regole red-black non sono rispettate da qualche parte nel codice (invece che scoprirlo in qualche caso poco frequente a runtime).

**Esempi di GADTs**

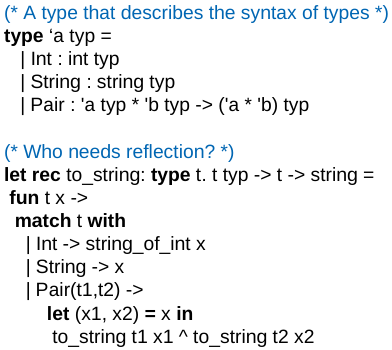
*Reflection* in Java

Un linguaggio con **reflection** è un linguaggio in cui a runtime è possibile estrarre informazioni di tipaggio da un dato. Ad esempio nel caso delle classi ottenere una rappresentazione di quanti e quali metodi ha e quanti e quali campi possiede. Possono essere usati ad esempio per discriminare in base al tipo della funzione un comportamento da adottare.

Lo svantaggio è che vengono portate dietro molte informazioni sui tipi, appesantendo l’esecuzione.

Esempio: **to\_string**

Con i GADT è possibile fare a meno della reflection in quanto è possibile rappresentare questa informazione di tipo a runtime solo quando necessaria.



Prima di tutto si introduce un tipo di dato ‘a typ che a runtime rappresenta un’informazione di tipo alpha (ad esempio l’atomo Int (solo atomo senza valori in input) avrà rappresentazione a runtime int typ, ecc). Qui “type t. t typ …” viene scritto a mano dal programmatore in base a cosa vuole rappresentare.

Per chiamare to\_string bisogna usare ad esempio: to\_string Int 4, oppure to\_string String “hello”.

La funzione to\_string (simile alla printf) ha come tipo: per ogni tipo t prendi un typ (con tipo) t, un elemento di t e restituisce una stringa. Senza il -> t -> (che rappresenta il vero e proprio tipo int, string, …, che è diverso dal tipo creato int typ, string typ, ecc) non sarebbe rappresentabile a runtime perché una stringa è una sequenza di bit (potrebbe essere un intero, una stringa, ecc), non viene mantenuta l’informazione sul tipo a runtime (di default), per questo è necessario passarla.

Si passa a to\_string quindi sia la rappresentazione sotto forma di tag del tipo di t (ovvero t typ) che t. Si fa poi pattern matching sulla rappresentazione del tipo, se ad esempio t è intero (quindi è come dire int typ) si può stampare x (elemento da stampare passato alla funzione) considerandolo un intero.

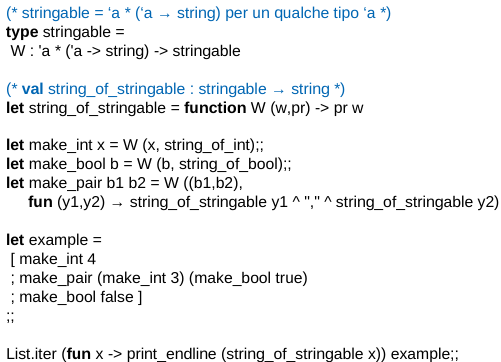
Per quanto riguarda Pair(t1, t2) si richiede che x sia nella forma ‘a \* ‘b quindi è possibile fare un pattern matching (let (x1, x2) = x), così il compilatore sa che è una coppia, e x1 avrà tipo t1 e x2 avrà tipo x2, chiamando ricorsivamente la stampa.

Il poliformismo uniforme viene rappresentato dal “**per ogni**”, tipi dei template, dei generics, di ML, … Ovvero quantificare sui tipi, per ogni alpha e beta succede qualcosa e quella funzione applicarla ad elementi di qualunque tipo.

Il duale è “**esiste**”, ad esempio esiste un tipo alpha per cui vale qualcosa. Corrisponde a nascondere l’implementazione del tipo. Come beneficio ha che è possibile cambiare l’implementazione di una parte senza che il codice subisca modifiche da altre parti. La nozione corrispondente all’esistenziale è data dal tipo di dato *astratto* (non algebrico).

Con i GADT è possibile scrivere tipi *esistenziali*.

Esempio: **esistenziali**



Qui W (w, pr) non è nome funzione e argomento, ma può essere vista come una funzione anonima che prende in input un elemento tag-valore in cui il tag è W e il valore una coppia (w, pr) questo fa si che l’input della funzione sia uno stringable (W non è una variante polimorfa quindi va definito prima di usarlo), e lo stesso vale anche per i W (x, string\_of) successivi, fanno tutti riferimento al W dichiarato nel variante stringable sopra.

Un elemento stringable altro non è che un dato di qualsiasi tipo e un modo per stamparlo.

La variabile string\_of\_stringable sarà rappresentata nello heap come tre celle contigue contenenti un atomo ‘W’, una cella con un dato di qualche tipo e un puntatore ad un chiusura in grado di stampare w.

Avendo le liste tutte lo stesso tipo in example si nascondono tutti i tipi all’interno di elementi stringable, questo impedirà in futuro di recuperare il valore originale (4 ad esempio) ma permetterà sempre di stamparlo.

In Java questo esempio potrebbe essere riprodotto definendo un’interfaccia Stringable con un unico metodo che restituisce una stringa, i make\_int, make\_bool, ecc sarebbero classi che implementano Stringable in cui ognuna avrebbe un campo inizializzato dal costruttore e il metodo toString() andrebbe a chiamare il metodo string\_of\_bool, string\_of\_int, ecc.

A runtime in example si avrebbero oggetti, quindi puntatori, e la rappresentazione sarebbe quindi più pesante. Inoltre con un’interfaccia si perde molta espressività. Si può inoltre effettuare un cast di un elemento della lista a runtime, o fare comunque un check del tipo e si scoprirebbe di che tipo è, cosa che non dovrebbe essere possibile per gli esistenziali.

Integrazioni alla spiegazione degli esistenziali:

Dato il tipo **print: ‘α \* ‘α → string → stringable**

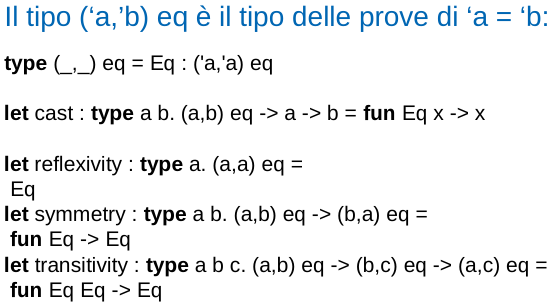
(ovvero una coppia formata da un elemento di un certo tipo α e una funzione per stamparlo, etichettata con l’atomo print in memoria è un printable) Qui c’è un ∀‘α. implicito all’inizio, che sta all’inizio della freccia, ovvero può essere visto come (l’implicazione è in realtà un se e solo se):

**print: (∀‘α . ‘α \* ‘α → string) ↔ stringable**

Questo tipo può essere scritto anche come esistenziale:

**print: ∃‘α.(‘α \* ‘α → string) ↔ stringable**

Esempio: **dimostrazioni**



Il tipo di dato (‘a, ‘a) eq sarà il tipo delle prove di alpha = beta. Esiste (a, b) eq per ogni a e b, ma è abitabile con eq, ovvero riesco a trovare un dato eq di a b, solamente se a e b sono identici (‘a, ‘a). Questo è chiamato *principio di riflessività*.

OCaml non ha cast, perché permetterebbe di dire che un tipo a ha poi tipo b, poi andrebbe portata l’informazione di tipo a runtime appesantendo l’esecuzione.

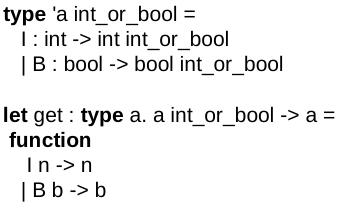
Attraverso il tipo di dato eq è possibile introdurre una funzione di cast:

let cast : type a b. (a,b) eq -> a -> b = fun Eq x -> x

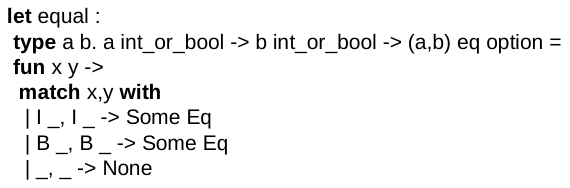
La funzione cast indica: per ogni tipo a e b, se si ha la dimostrazione che a è uguale a b, allora puoi ottenere una funzione da a in b. Questa funzione è fornita solo sotto previa dimostrazione in input che a sia uguale a b. la funzione usa Eq che vale solamente quando a e b sono uguali, quindi è possibile ritornare l’elemento passato in input come output.

Seguono dimostrazioni di riflessività (dimostrare che a è uguale ad a), simmetria (dimostrare che se a è uguale a b allora anche b è uguale ad a) e transitività (se a è uguale a b e b è uguale a c allora anche a è uguale a c).

Esempio: **safe cast** a runtime

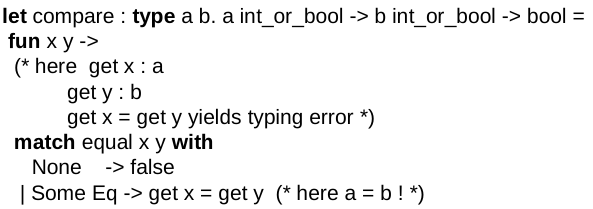


Si ha un tipo (GADT) int\_or\_bool e una funzione get che restituisce int o bool. Se aggiungiamo la funzione equal che dati in input due int\_or\_bool calcoli se sono uguali oppure no:



Dati due elementi ritorna un Eq option che sarà Some se sono uguali, altrimenti None. Quindi viene ritornato non solo se a e b sono uguali ma se lo sono anche la dimostrazione.

Altro esempio è la *compare*:



In conclusione i GADT permettono forme di dimostrazione limitate:

- forzare invarianti dei dati e usarli per semplificare il codice

- proprietà dei dati come oggetti di prima classe (e.g. il tipo (‘a,’b) eq)

- caso particolare: tipi di dati esistenziali (già codificati in Hindley-Milner, essenziali per il data hiding).