Generalized algebraic data types

L'idea di base è che i tipi possono dipendere da termini. Alcuni esempi possono essere:

- Liste di lunghezza n (`list n`).
- Coppie formate da uno stato e da una città di quello stato (`country:Country * City(country)`).
- Funzioni che dato un `n` restituiscono una lista lunga `2n` (`n:nat -> list(2*n)`).
- Funzioni che data una lista ritornano una lista che ne è una permutazione (`l1:  list -> l2:list * is_permutation l1 l2`).

In generale, dato un enunciato `P`, possiamo rappresentare con un tipo di dato tutte le prove di `P`.

Quindi in un linguaggio con tipi dipendenti possiamo scrivere specifiche esatte (vedi esempio permutazione di liste).

Si può quindi affermare che:

- Programmare quindi diventa abitare tipi, oppure scrivere dimostrazioni.
- Passare alle funzioni input diventa passare prove (ad esempio, che l'input soddisfa le precondizioni).
- La programmazione diventa dimostrazione interattiva.

Va però detto che lasciare che il compilatore fornisca le prove per noi è impossibile (il problema è indecidibile).

I linguaggi di programmazione scelgono un tradeoff fortemente sbilanciato sulla correttezza del codice rispetto a semplicità d'uso e di compilazione.

Questo concetto non è ancora pienamente emerso perché probabilmente troppo complesso, e con un rapporto costi/benefici troppo sbilanciato sui costi.

Si osservi ora un esempio di **ADT NON generalizzati** in Python.
Si vuole rappresentare con una struttura dati gli alberi di sintassi astratta delle espressioni di un linguaggio di programmazione.

```
from __future__ import annotations
from dataclasses import dataclass

@dataclass
class Num:
    n: int

@dataclass
class String:
    n: str

@dataclass
class Mult:
    e1: Expr
    e2: Expr

@dataclass
class Concat:
    e1: Expr
    e2: Expr

@dataclass
class Eq:
    e1: Expr
    e2: Expr

Expr = Num | String | Mult | Concat | Eq
```

`dataclass` è un generatore di codice, che analizza e genera il codice boilerplate delle classi. `Expr = Num | String | Mult | Concat | Eq` indica che la nostra `Expr` deve avere uno di quei tipi.

Per esprimere lo stesso concetto in OCaml:

```
type expr =
    Num : int -> expr
  | Bool : bool -> expr
  | Mult : expr * expr -> expr
  | And : expr * expr -> expr
  | Eq : expr * expr -> expr
```

Questa struttura dati è interessante perché in molti casi vogliamo avere una struttura dati che mantenga delle invarianti (per invariante si intende l'insieme delle proprietà che la struttura dati deve rispettare, ad esempio gli Alberi Red & Black con le loro regole).

L'invariante da aggiungere a questa struttura è che solo alcune di queste espressioni hanno un senso. L'espressione deve essere *ben tipata*.

Si osservino due esempi di espressioni, una corretta ed una errata (senza un senso per i tipi appena dichiarati), in Python:

```
good = Eq(Mult(Num(3), Num(4)), Num(5))

bad = Eq(Concat(String("ciao"), Num(2)), Num(3))
```

La seconda espressione (`bad`) non ha un senso logico dati i tipi dichiarati in precedenza, ma con il codice attuale non verrebbe sollevato alcun errore (non ci sono controlli sui tipi).

La prima espressione soddisfa l'invariante *rappresentare una espressione corretta e di tipo intero*, la seconda no.

Si vuole ora implementare una funzione che valuti le nostre espressioni:

```
def eval(e):
    match e:
        case Num(n):
            return n
        case String(s):
            return s
        case Mult(e1, e2):
            return eval(e1) * eval(e2)
        case Concat(e1, e2):
            return eval(e1) + eval(e2)
        case Eq(e1, e2):
            return eval(e1) == eval(e2)
```

Non viene tipata la funzione `eval` perchè non sarebbe tipabile con questo codice. Ad esempio, il primo ed il secondo caso restituirebbero tipi differenti.

Volendo quindi valutare le due espressioni precedenti:

```
good = Eq(Mult(Num(3), Num(4)), Num(5))
bad = Eq(Concat(String("ciao"), Num(2)), Num(3))

print(f"Good: {eval(good)}")
print(f"Bad: {eval(bad)}")
```

Non verrebbe sollevato alcun errore prima dell'esecuzione. Durante l'esecuzione la prima stampa verrebbe effettuata (con esito `false`), ma la seconda espressione andrebbe a sollevare un errore (non si può concatenare una stringa con un intero).

Questo è un caso molto frequente nell'informatica, dove non si riesce a tipare la funzione visti i diversi tipi possibili di ritorno. Anche con un tipo di ritorno disgiunto (`int | str | bool`) il problema non sarebbe risolto.

In questo caso, il sistema di tipi di Python permette di non tipare la funzione, a differenza di linguaggi (come OCaml) che pretendono un tipo di ritorno.

Si osservi proprio lo stesso codice in OCaml (riprendendo la definizione data già in precedenza in OCaml per la struttura dati):

```
type res =
    | I : int -> res
    | B : bool -> res
    | E : res

let to_string =
    function
        I n -> string_of_int n
    |   B b -> string_of_bool b
    |   E -> "error"

let rec eval: expr -> ??? =
    function
        Num n -> I n
    |   Bool b -> B b
    |   Mult(e1, e2) ->
            (match eval e1, eval e2 with
             | I n1, I n2 -> I (n1 * n2)
             | _, _ -> E)
    |   And(e1, e2) ->
            (match eval e1, eval e2 with
             | B b1, B b2 -> B (b1 && b2)
             | _, _ -> E)
    |   Eq(e1, e2) ->
            (match eval e1, eval e2 with
             | I n1, I n2 -> B (n1 == n2)
             | B b1, B b2 -> B (b1 == b2)
             | _, _ -> E)

let good = Eq (Mult (Num 3, Num 4), Num 5)
let bad = Eq (And (Bool true, Num 4), Num 5)

let _ =
    Printf.printf "%s\n" (to_string (eval good)) ;
    Printf.printf "%s\n" (to_string (eval bad))
```

Senza i GADT, il codice risulta essere molto complesso e verboso, dovendo controllare diversi casi e dovendo implementare un tipo di risposta che permette di restituire interi, booleani oppure un errore.

Eseguendo questo codice, il risultato sarebbe `false` ed `error`.
Il codice viene però appesantito rispetto al codice Python visto in precedenza.

Si introducono ora i **GADT**.
Si vogliono spiegare al compilatore, tramite predicati sulla struttura dati, le invarianti che vogliamo dare alla struttura dati.

Il codice per la dichiarazione della struttura in OCaml diventa:

```
type 'a expr =
    Num : int -> int expr
  | Bool : bool -> bool expr
  | Mult : int expr * int expr -> int expr
  | And : bool expr * bool expr -> bool expr
  | Eq : (* forall 'b *) 'b expr * 'b expr -> bool expr
```

`'a` indica il tipo di ritorno della funzione. Può essere visto come un costruttore di tipo.
Ogni espressione può avere, in base alla necessità, un tipo di ritorno differente. Il tipo dipende dal tipo dell'input.

Provando ora ad eseguire la valutazione delle due espressioni precedenti, l'espressione `bad` restituirebbe un errore. Viene quindi codificata l'invariante all'interno della definizione della struttura, permettendo al compilatore di controllare la validità dei tipi.

Va ora modificato il codice di `eval`:

```
let rec eval: type.a a expr -> a =
    function
        Num n -> n
    |   Bool b -> b
    |   Mult(e1, e2) -> eval e1 * eval e2
    |   And(e1, e2) -> eval e1 && eval e2
    |   Eq(e1, e2) -> eval e1 == eval e2
```

Si può quindi rimuovere il tipo `res` custom che prima era necessario.
Il tipo di ritorno diventa proprio `'a`.
`type a.  a expr -> a` permette di avere polimorfismo.

Il codice è molto simile a quello in Python, più semplice e leggibile, con la differenza che l'espressione `bad` non è eseguibile (in Python si).

Bisogna quindi chiarire come fa il compilatore a tipare la funzione.

Nei linguaggi con GADT (ma anche negli if-then-else come in Kotlin) **il compilatore può riconoscere determinate forme del codice** (il pattern matching per i GADT, determinate guardie negli if-then-else in Kotlin).

Quando la forma viene riconosciuta, il compilatore introduzione **equazioni fra tipi**.
Le equazioni vengono quindi risolte, potendo determinare il **cambio del tipo di variabili**, **tipi di ritorno**, etc. (in Kotlin solo le varibili cambiano tipo, e l'unica variazione di tipo può essere lungo la gerarchia di ereditarietà oppure da nullable a tipo non nullable).

Si osservi quindi come ragiona il compilatore di OCaml, riprenendo il codice di `eval` precedente ed aggiungendo commenti per ogni caso specifico:

```
let rec eval: type.a a expr -> a =
 (* Il tipo dell'input è              a expr *)
 (* Il tipo da dare in output è        a      *)
    function
        (*  Siamo nel caso Num! Num è dichiarato come segue:
            Num : int -> int expr
            Quindi:     n : int
            L'input (Num n) ha tipo     int expr
            Quindi:     a expr = int expr
            Per iniettività dei costruttori di GADTs:   a = int
            Quindi: il tipo di ritorno che era a ora è int!         *)
        Num n -> n
        (*  Siamo nel caso Bool! Bool è dichiarato come segue:
            Bool : bool -> bool expr
            Quindi:     b : bool
            L'input (Bool b) ha tipo     bool expr
            Quindi:     a expr = bool expr
            Per iniettività dei costruttori di GADTs:   a = bool
            Quindi: il tipo di ritorno che era a ora è bool!        *)
    |   Bool b -> b
        (*  Siamo nel caso Mult! Mult è dichiarato come segue:
            Mult : int expr * int expr -> int expr
            Quindi:     e1 : int expr, e2 : int expr
            L'input (Mult(e1, e2)) ha tipo     int expr
            Quindi:     a expr = int expr
            Per iniettività dei costruttori di GADTs:   a = int
            Quindi: il tipo di ritorno che era a ora è int!
            Inoltre: la eval ha tipo    'a expr -> 'a
            Quindi: eval e1 ha tipo int, eval e2 ha tipo int        *)
    |   Mult(e1, e2) -> eval e1 * eval e2
    |   And(e1, e2) -> eval e1 && eval e2
        (*  Siamo nel caso Eq! Eq è dichiarato come segue:
            Eq : forall 'b. 'b' expr * 'b' expr -> bool expr
            Sia 'b un tipo ignoto ma fissato
            Quindi:     e1 : 'b' expr, e2 : 'b' expr
            L'input (Eq(e1, e2)) ha tipo     bool expr
            Quindi:     a expr = bool expr
            Per iniettività dei costruttori di GADTs:   a = bool
            Quindi: il tipo di ritorno che era a ora è bool!
            Inoltre: la eval ha tipo    'a expr -> 'a
            Quindi: eval e1 ha tipo 'b', eval e2 ha tipo 'b'        *)
    |   Eq(e1, e2) -> eval e1 == eval e2
```

Il compilatore riesce quindi autonomamente a risolvere le equazioni fra tipi, determinando il tipo di ritorno.

Vengono non solo catturate le invarianti, semplificando il codice, ma grazie ai GADT possiamo modificare il tipo di ritorno in base al tipo in input.
Questo è molto utile in tantissimi casi comuni, come la `printf`, il tipo di ritorno delle query a database, il tipare i pacchetti di rete in base al primo pacchetto ricevuto, etc.

Si vuole ora provare ad implementare la funzione `printf` presente in C, utilizzando i GADT.

```
(* Esempio di una funzione printf-like:
        val to_string : 'a spec -> 'a -> string *)

(* Le specifiche corrispondono ai vari %.. della printf *)
type 'a spec =
  | D : int spec
  | F : float spec
  | S : string spec
  | Pair : 'a spec * 'b spec -> ('a * 'b) spec
  | List: 'a spec -> ('a list) spec

let rec to_string : type a. a spec -> a -> string =
    function
        D -> string_of_int
     |  F -> string_of_float
     |  S -> (fun x -> x)
     |  Pair(s1, s2) ->
            (*  Poichè l'input è Pair(s1, s2):
                s1 : 'a spec
                s2 : 'b spec
                a spec = ('a * 'b) spec
                Quindi      a = 'a * 'b
                e quindi devo restituire una funzione
                            'a * 'b -> string
            *)
        (function (x,y) ->      (* x : 'a       y : 'b *)
            "<" ^ to_string s1 x ^ "," ^ to_string s2 y ^ ">")
    | List(s) ->
        (function l ->
            "[" ^ String.concat "," (List.map (to_string s) l) ^ "]")

let _ =
    let example_spec = List (Pair (D,F)) in
    print_string (to_string example_spec [(3,3.5);(0,1.14)])
```

Questo codice restituisce una stringa $[< 3, 3.5 >, < 0, 1.14 >]$. Viene quindi implementata in user space una `printf` sotto steroidi.

A compile time viene fatto il pattern matching e viene risolta l'equazione dei tipi.
Ma cosa succede a runtime?

A runtime il discorso sui tipi viene cancellato. Il tipo da stampare è una sequenza di bit. Non è presente alcun overhead. Il primo argomento passato è quello che determina il tipo di ritorno.

I GADT permettono di fare cose che normalmente non si potrebbero fare facilmente, senza litigare troppo col compilatore. Permettono infatti di aiutare il compilatore.

Si vuole ora osservare un esempio, riprendendo la seguente dichiarazione vista in precedenza:

```
type 'a expr =
    Num : int -> int expr
  | Bool : bool -> bool expr
  | Mult : int expr * int expr -> int expr
  | And : bool expr * bool expr -> bool expr
  | Eq : (* forall 'b *) 'b expr * 'b expr -> bool expr
```

Si vuole scrivere una funzione che date due espressioni mi dica se sono uguali o diverse (da utilizzarsi nel parses di un compilatore):

```
let equal : type a b. a expr -> b expr -> bool =
    function e1 e2 -> e1 == e2
```

Questa prima definizione non compila. In OCaml (come in Erlang) le informazioni sui tipo vengono perse a runtime. Si possono quindi confrontare solo espressioni dello stesso tipo (anche se la rappresentazione in bit è la stessa).

Serve quindi capire se le due espressioni hanno lo stesso tipo, per *castare* una delle due ed effettuare il confronto.

Si scrive quindi una funzione per controllare se le due espressioni hanno lo stesso tipo:

```
let same_type : type a b. a expr -> b expr -> bool =
    fun e1 e2 ->
        match e1, e2 with
         |  (Num _ | Mult _), (Num _ | Mult _) -> true
         |  (Bool _ | And _ | Eq _), (Bool _ | And _ | Eq _) -> true
         |  _, _ -> false
```

con conseguente funzione per effettuare il confronto tra le due espressioni:

```
let equal : type a b. a expr -> b expr -> bool =
    fun e1 e2 ->
        if same_type e1 e2 then
            e1 = e2
        else
            false
```

Per far accettare al compilatore l'espressione `e1 = e2` si dovrebbe implementare una funzione `cast` (non presente in OCaml) per castare una delle due espressioni nel tipo dell'altra.

In generale, il casting non è una buona idea. Vogliamo comunque implementare una sorta di cast per risolvere il nostro problema.

Per fare ciò vogliamo utilizzare le dimostrazioni. Programmare è come dimostrare enunciati matematici. Se il sistema di tipi è abbastanza espressivo, certi tipi corrispondono a certi enunciati matematici.
Un dato di un certo tipo corrisponde ad una dimostrazione. Si può prendere in input una dimostrazione per dire al compilatore che un qualcosa è vero.

Si vuole quindi ora modificare il nostro codice, utilizzando dimostrazioni per convincere il compilatore a castare il tipo delle espressioni.

Si osservi quindi un nuovo tipo `dimostrazione eq` per l'uguaglianza:

```
type ('a, 'b) eq =
  | Refl : (* forall 'c *) ('c, 'c) eq
```

Viene utilizzata la dimostrazione di uguaglianza che afferma: *l'uguaglianza è il più piccolo predicato riflessivo* $(x = x)$.

Dimostriamo ora le proprietà dell'uguaglianza:

```
(* Dimostrazione della riflessività *)
let refl : type a. (a, a) eq = Refl
```

```
(* Dimostrazione della simmetria *)
let symm : type a b. (a, a) eq -> (b, b) eq =
    function
        (*  L'input è Refl tipato con forall 'c. ('c, 'c) eq
            Sia 'c un tipo fissato
            Quindi      (a, b) eq = ('c, 'c) eq
            Quindi      a = 'c = b
            Quindi il mio output diventa
                (b, a) eq   ovvero  ('c, 'c) eq
        *)
        Refl -> Refl
```

```
(* Dimostrazione della transitività *)
let trans : type a b c. (a, b) eq -> (b, c) eq -> (a, c) eq =
    fun x y ->
        match x, y with
         |  Refl, Refl -> Refl
```

Si può quindi modificare il codice di `same_type`, modificando il tipo di ritorno:

```
(* Restituisco: None se non hanno lo stesso tipo,
                Some Refl se hanno lo stesso tipo *)
let same_type : type a b. a expr -> b expr -> (a expr, b expr) eq option=
    fun e1 e2 ->
        match e1, e2 with
         |  (Num _ | Mult _), (Num _ | Mult _) -> Some Refl
         |  (Bool _ | And _ | Eq _), (Bool _ | And _ | Eq _) -> Some Refl
         |  _, _ -> None
```

Nota Bene: i casi del match andrebbero scritti singolarmente, ma la logica è uguale!

Viene quindi modificato anche il codice di `equal`, con conseguente implementazione di `cast`:

```
let cast : type a b. (a, b) eq -> a -> b =
    function
        (*  Poichè Refl : ('c, 'c) eq
            si ha   (a, b) eq = ('c, 'c) eq
            quindi  a = b = 'c
            e il tipo di ritorno a -> b diventa 'c -> 'c
        *)
        Refl -> (fun x -> x)

let equal : type a b. a expr -> b expr -> bool =
    fun e1 e2 ->
        match same_type e1 e2 with
            Some p -> e1 = cast (symm p) e2
        else
            None -> false
```

Nota Bene: il cast non altera in nessun modo la memoria!
Cast cambia il tipo che si associa al dato, ma il dato rimane lo stesso.

Per riassumere, abbiamo effettuato grazie ai GADT i check necessari per effettuare il cast, dando in pasto al compilatore OCaml le dimostrazioni necessarie per convincerlo.

Il cast a runtime non ha alcun costo, grazie alle ottimizzazioni del compilatore (traducendo pattern matching in singole espressioni, se possibile).