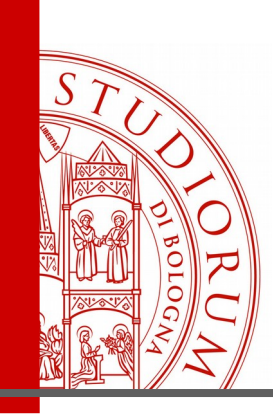


Generalized Algebraic Data Types

Tipi Dipendenti

Idea: i tipi possono dipendere da termini

Esempi:

- 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

Tipi Dipendenti

Idee:

- 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)
- programmare = abitare tipi = scrivere dimostrazioni

Tipi Dipendenti

Idee:

- passare alle funzioni input = passare prove (p.e. che l'input soddisfa le precondizioni)
- programmazione = dimostrazione interattiva
- lasciare che il compilatore fornisca le prove per noi è impossibile (il problema è indecidibile)

Tipi Dipendenti

Sistemi:

- Dimostratori interattivi di teoremi: Agda, Coq, Lean, Matita, NuPRL, PVS, Twelf, …
- Linguaggi di programmazione: ATS, Cayenne, Dependent ML, F*, Idris, …

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

Tipi Dipendenti

Un concetto non (ancora?) emerso:

- troppo complessi
- rapporto costi/benefici troppo sbilanciato sui costi

Casi particolari di tipi dipendenti:

- tentativi di immettere tipi dipendenti a piccole dosi, mantenendo il tipaggio decidibile (o quasi) senza chiedere interventi dell'utente (prove esplicite)
- Es.: tipi dipendenti da naturali con + e prodotti scalari (es. liste di lunghezza n)

Generalized ADT (GADT)

Un concetto (lentamente) emergente:

- proposti nel 1994 come semplificazione e contemporanea generalizzazione dei tipi di dati induttivi di Coq et alt.
- adattati a Haskell/ML nel 2003 e combinati con altre estensioni di Haskell nel 2006
- in OCaml dal 2012
- proposti/parzialmente implementati/simulabili a fatica in Scala, F#, ...

GADT: perché?

Sintassi delle espressioni come ADT:

```
type expr =
    Num of int
  | Bool of bool
  | Plus of expr * expr
  | And of expr * expr
  | Eq of expr * expr
```

```
let so_bad =
   Plus (Bool true) (Eq
(Num 3) (Bool false))
```

so_bad è sintatticamente corretta, ma non rappresenta un'espressione ben tipata

GADT: perché?

Sintassi delle espressioni come ADT:

```
type expr =
    Num of int
  | Bool of bool
  | Plus of expr * expr
  | And of expr * expr
  | Eq of expr * expr
```

```
(* eval : expr → nat_or_bool
    raises BadArgs when ... *)
let rec eval =
 function
    Num n →  I n
  | Bool b  → B b
  | Plus(e1,e2) →
     (match eval e1, eval e2 with
          I n, I m → I (n+m)
        | _, _ → raise BadArgs)
  | ...
```

GADT: perché?

Sintassi delle espressioni come GADT:

type ‘a expr =
  Num **:** int → int expr
 | Bool **:** bool expr
 | Plus **:** int expr * int expr → int expr
 | And : bool expr * bool expr → bool expr
 | Eq **:** ‘a expr * ‘a expr → bool expr

I costruttori istanziano i parametri del tipo!

Es. Plus (Num 3) (bool false) è mal tipato

GADT: perché?

```
type ‘a expr =
   Num : int → int expr
 | Bool : bool expr
 | Plus : int expr * int expr → int expr
 | And : bool expr * bool expr → bool expr
 | Eq : ‘a expr * ‘a expr → bool expr
```

```
(* val eval : 'a expr -> 'a *)
let rec eval : type a. a expr → a =
 function
    Num n → n    (* here Num n : int expr and so a = expr *)
  | Bool b → b
  | Plus(e1,e2) → eval e1 + eval e2
  | And(e1,e2) → eval e1 && eval e2
  | Eq(e1,e2) → eval e1 = eval e2
```

GADT

- un GADT è isomorfo a run-time all'ADT ottenuto cancellando i parametri istanziati dai costruttori

Dichiarazione:	A run-time:
Plus of expr * expr	{ Plus, e1, e2 }
Plus : int expr * int expr → int expr	{ Plus, e1, e2 }

- l'analisi per casi su un elemento di un GADT introduce nuovi vincoli tenendo conto del tipo più preciso dei costruttori

| Plus(e1,e2) → …. (* expr int is the type of the scrutinee*)

GADT

- I vincoli di tipaggio permettono di imporre invarianti sui dati. Esempi:
  - (‘a expr) rappresenta espressioni ben tipate
  - (‘a channel) è un canale di sola lettura se ‘a = read, di sola scrittura se ‘a = write, etc.
  - (‘a red_black_node) è un nodo di colore ‘a di un red-black tree
  - ...
- il codice ottenuto è più breve e corretto: non deve trattare i casi che violano gli invarianti

GADT

- in presenza di GADT la Hindley-Milner type inference non è più decidibile e si perde most-general unifier
- per ovviare l'utente talvolta inserisce annotazioni di tipo per aiutare il compilatore

(* **val** eval : 'a expr → 'a *)

let rec eval : **type** a. a expr → a = …

- la differenza fra 'a e **type** a è subdola: afferma che a non può essere reso meno generale di 'a, tranne che da parte dei vincoli imposti dai GADT

GADTs: esempio

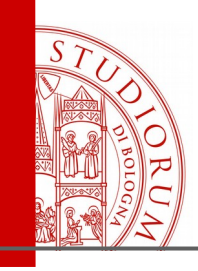
```
(* A type that describes the syntax of types *)
type ‘a typ =
   | Int : int typ
   | String : string typ
   | Pair : 'a typ * 'b typ -> ('a * 'b) typ

(* Who needs reflection? *)
let rec to_string: type t. t typ -> t -> string =
 fun t x ->
  match t with
    | Int -> string_of_int x
    | String -> x
    | Pair(t1,t2) ->
        let (x1, x2) = x in
         to_string t1 x1 ^ to_string t2 x2
```

GADTs: esempio

```
(* stringable = ‘a * (‘a → string) per un qualche tipo ‘a *)
type stringable =
 W : 'a * ('a -> string) -> stringable

(* val string_of_stringable : stringable → string *)
let string_of_stringable = function W (w,pr) -> pr w

let make_int x = W (x, string_of_int);;
let make_bool b = W (b, string_of_bool);;
let make_pair b1 b2 = W ((b1,b2),
    fun (y1,y2) → string_of_stringable y1 ^ "," ^ string_of_stringable y2)

let example =
 [ make_int 4
 ; make_pair (make_int 3) (make_bool true)
 ; make_bool false ]
;;

List.iter (fun x -> print_endline (string_of_stringable x)) example;;
```

GADT: esempio

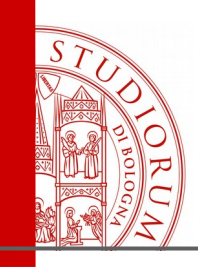
- W (4, string_of_int) viene rappresentato a run-time come {W, 4, fun string_of_int/1 }
- ovvero: il tipo di 4 NON viene rappresentato a run-time
- confronta con
  - Oggetti/interfacce: come implementereste il codice precedente tramite oggetti?
  - Tipi taggati: W_int **of** int * (int → string)
  - Puntatori alla classe come tag (negli oggetti)



GADT: esempio

Il tipo (‘a,’b) eq è il tipo delle prove di ‘a = ‘b:

```
type (_,_) eq = Eq : ('a,'a) eq

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

let reflexivity : type a. (a,a) eq =
 Eq
let symmetry : type a b. (a,b) eq -> (b,a) eq =
 fun Eq -> Eq
let transitivity : type a b c. (a,b) eq -> (b,c) eq -> (a,c) eq =
 fun Eq Eq -> Eq
```



GADT: esempio

Permette cast di tipo safe a run-time:

```
type 'a int_or_bool =
   I : int -> int int_or_bool
   | B : bool -> bool int_or_bool

let get : type a. a int_or_bool -> a =
 function
    I n -> n
   | B b -> b

...
```

GADT: esempio

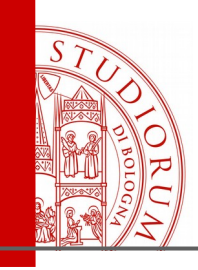
```
let equal :
 type a b. a int_or_bool -> b int_or_bool -> (a,b) eq option =
 fun x y ->
  match x,y with
   | I _, I _ -> Some Eq
   | B _, B _ -> Some Eq
   | _, _ -> None
```

GADT: esempio

```
let compare : type a b. a int_or_bool -> b int_or_bool -> bool =
 fun x y ->
  (* here  get x : a
           get y : b
           get x = get y yields typing error *)
  match equal x y with
     None    -> false
   | Some Eq -> get x = get y  (* here a = b ! *)
```

GADT: conclusioni

- un costrutto che sta lentamente emergendo
- permette forme limitate di dimostrazione:
  - forzare invarianti dei dati e usarli per semplificare il codice
  - proprietà dei dati come oggetti di prima classe (p.e il tipo (‘a,’b) eq)
  - caso particolare: tipi di dati esistenziali
    - già codificabili in Hindley-Milner
    - essenziali per data hiding



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it