

Sommario: 22 aprile, 2021

- Dati, Tipi di Dato e Tipi
- Tipi: Principi e Usi
- Sistemi di Tipi e Type Safety
- Equivalenza, Coercion and Cast
- Polimorfismo: Ad Hoc, Generico e di Sottotipo
- PreSAI: Una metodologia per Nuovi Tipi di Dato
- Studio di Casi: Tipi in C e il costruito Union
- Studio di Casi: Tipi algebrici (o concreti) in OCaml
- Espressività
- Esercizi

- **Dato:** La più semplice struttura **per introdurre valori** in un programma
- **Caratteristiche Distintive** sono le differenti modalità con cui possono essere introdotti ed usati nei programmi:
  - Valori Calcolabili: Introdotti da espressione non atomica;
  - Valori Denotabili: Introdotti come Denotazioni nei Frame;
  - Valori Modificabili: Valori con Stato;
  - Valori Memorizzabili: Valori associato allo Stato di un modificabile;
  - Valori Literals: Atomici.
- Ogni Linguaggio definisce quali modi sono usabili per i dati

## Example

```
int A[] = {2, 7, 12, 9}; /* corretto in C * /  
intSeq QSort(intSeq cc, int ord(int x, int y)){...}; /* errore in C90 * /  
A = {2, 7, 12, 9}; /* errore in C * /
```

- {2,7,12,9} esprime in C, un valore array solo nella dichiarazione
- una funzione in C90 è solo un valore denotabile: non può essere trasmessa, nè restituita, nè assegnata come valore memorizzabile

# Dati, Tipi di Dato, Tipi: Tipi di Dato

- **Tipi di Dato: Collezioni di valori omogenei** rispetto alle operazioni che possono essere loro applicate, nei programmi
  - **Caratteristiche Distintive** sono le operazioni applicabili
  - Una grande varietà. Si suddividono in:
    - **Scalari:** Sono valori atomici  
Esempi: *Interi, Virgola mobile, Complessi, Caratteri, Boolean, Enumerato, Intervallo, Void, Puntatore, ...*
    - **Strutturati:** Sono valori composti con operazioni per la selezione dei componenti  
Esempi: *Record, Array, Stringhe, Set, List, Collections, ...*
      - **Statici:** Sono allocati staticamente, alla dichiarazione/introduzione
      - **Dinamici:** Sono allocati dinamicamente mediante espressioni che ne estendono o riducono la struttura dei componenti
      - **Ricorsivi:** Sono tipi definiti in modo induttivo richiedono uno specifico meccanismo: Emulato in C mediante puntatori  
Esempi. `Type intList = Null | Cons int intList`
  - **Memorizzabili, Modificabili o no:** Il linguaggio stabilisce
    - Per gli scalari, quali siano, oppure no, assegnabili a variabili (ovvero quali siano memorizzabili)
    - Per gli strutturati, quali abbiano componenti modificabili.

# Dati, Tipi di Dato, Tipi: Tipi

- **Tipi: Strutture per classificare in modo univoco** il "ruolo" di ogni struttura usata in un programma con vari effetti e scopi:
  - **Evidenziare** l'organizzazione concettuale del programma  
Esempi. La presenza nel programma (a top-level<sup>1</sup>) di tipi per dati e operazioni che rispecchiano l'organizzazione dei valori e delle trasformazioni operate dal problema da meccanizzare. Ad esempio: Il programma deriva introduce il tipo `parseTree`.
  - **Supportare il controllo** dell'allocazione di memoria dei dati  
Esempi. È il caso della funzione `malloc` in C e `new` in altri linguaggi, che usano i tipi per calcolare l'ammontare di memoria dinamica da allocare. Analogamente, il compilatore guarda ai tipi, nelle dichiarazioni, per la memoria statica.
  - **Verificare** proprietà statiche del programma
  - **Type Safety** ovvero Verifica contro stati di "stuck"

---

<sup>1</sup> ma per i principi della Programmazione Strutturata, ciò si ripete ad ogni livello con i sotto-problemi

- **Tipi: Strutture per classificare in modo univoco** il "ruolo" di ogni struttura usata in un programma con vari effetti e scopi:
  - **Evidenziare** l'organizzazione concettuale del programma
  - **Supportare il controllo** dell'allocazione di memoria dei dati
  - **Verificare** proprietà statiche del programma  
Esempi. È il caso della corrispondenza tra le entità definite e il loro uso nel programma, quali variabili assegnate e valore assegnabile, funzione dichiarata e sue invocazioni, struttura di un valore usato e operazioni applicate ad esso, ...
  - **Type Safety** ovvero Verifica contro stati di "stuck". Uno stato di "stuck" è uno stato che nessuna corretta esecuzione del programma dovrebbe raggiungere<sup>2</sup>.  
Esempi. È il caso del seguente programma C.

---

<sup>2</sup> Uno stato di "stuck" non è uno stato con anomalie quali eccezioni di programma o sistema di varia origine. Tali stati sono invece da considerarsi legali e raggiungibili anche da computazioni corrette.

- **Type Safety** ovvero Verifica contro stati di "stuck". Uno stato di "stuck" è uno stato che nessuna corretta esecuzione del programma dovrebbe raggiungere<sup>3</sup>.

```
int main (int argc, char *argv[]){
    int x=10;
    char a = 'e';
    printf("Totale da pagare = %2d\n", x+'e');
    return 0;
}

/*
host-131-114-218-180:Programmi marcob$ cc stuck2.c -std=c90 -o stuck2c90.exe
host-131-114-218-180:Programmi marcob$ ./stuck2c90.exe
Totale da pagare = 111
host-131-114-218-180:Programmi marcob$
```

- Dovrebbe essere evidente che il programmatore ha commesso un errore
- Nondimeno, questa è la peggiore situazione che può capitare nella programmazione
  - Il programma è errato ma appare corretto e termina in uno stato che sembra corretto ma è uno "stuck" (lett. pantano)
  - Come riconoscerlo? Come Impedirlo: Usare un sistema di tipi (sound)

---

<sup>3</sup> Uno stato di "stuck" non è uno stato con anomalie quali eccezioni di programma o di sistema. Stati con eccezione sono invece da considerarsi legali e raggiungibili anche da computazioni corrette.

# Sistema di Tipi

- Consiste di 3 strutture:
  - **Dominio dei Tipi** (che include i tipi basici del Linguaggio)  
Esempi.  $T = \text{Void} + \text{Int} + \text{Char} + \dots$
  - (Linguaggio delle) **Espressioni di Tipo** con cui possono essere definiti tipi, anche derivati  
Esempi.  $T = \dots + T \times \dots \times T \rightarrow T + \dots$
  - **Regole (di inferenza)** con cui il sistema associa un unico tipo a ogni struttura del programma  
Esempi.  
$$\frac{E : \text{bool} \quad C : \text{Void}}{\text{while } E \text{ do } C : \text{Void}} \quad \frac{E : \text{bool} \quad E_{\text{then}} : T1 \quad E_{\text{else}} : T2 \quad T1 = T = T2}{E ? E_{\text{then}} E_{\text{else}} : T}$$
- Se il linguaggio ha Sistema di Tipi *sound*, allora

## Proposition

*Un programma è safe da "stuck" se il Sistema assegna a ogni struttura 1 e 1 solo tipo.*

## Esempio

*Il Sistema Y definito per Small21 è un esempio di sistema (da provare) sound per il Linguaggio Small21.*

# Relazioni e Proprietà sulla struttura dei Tipi

Intervengono quando **non c'è identità** tra tipo atteso e tipo effettivo.

Esempio.  $\{ \dots T1 \ x; \dots x = E; \dots \}$ . Il tipo atteso per  $E$  è  $T1$ , ma  $E$  ha tipo  $T2 \neq T1$ .

- **Equivalenza.** Sia  $x:T$  and  $y:T'$ . Quando  $x$  e  $y$  hanno stesso tipo?
  - **Nominal.** Ogni tipo è uguale solo a se stesso.  
Esempi. `{type T1 = int[5]; T1 x; int[5] y; x[0]=y[0]=12;}` contiene un errore di tipo
  - **Structural.** Stessa espressione di Tipo quando i nomi sono rimpiazzati dalle definizioni.  
Esempi. `{type T1 = int[5]; T1 x; int[5] y; x[0]=y[0]=12;}` sequenza ha tipo void
- **Compatibilità e Sottotipo**
  - $T$  è compatibile con  $S$  se (un valore di tipo)  $T$  può essere usato quando è atteso  $S$ .  
Esempi. Sottotipi  $T \subset S$  in Java:  
`type S = d_S; ...type T ⊂ S = d_T; ....T x = e_x; ....S y = x;`
- **Coercion:**  $T$  è compatibile con  $S$  ma i valori di  $T$  devono cambiare rappresentazione per essere usati come  $S$ . Esempi. `int` e `float` in C
- **Cast:** compatibilità senza necessità di cambiamenti sulla rappresentazione.  
Esempi. `...type T ⊂ S = d_T; ....T x = e_x; ....S y = x; ....; T z = (T)y;`

Intervengono quando **non c'è identità** tra tipo atteso e tipo effettivo.

Esempio.  $\{T1 \ x; \dots; \ x = E\}$ . Il tipo atteso per E è T1, ma E ha tipo  $T2 \neq T1$ .

- **Overloading** (Polimorfismo ad Hoc) Differenti tipi e valori funzionali per uno stesso identificatore di funzione  
Esempi. +, \*, ... per int e float hanno definizioni completamente diverse ma si chiamano nello stesso modo.
- **Parametric Polymorphism** (Polimorfismo generico) Le espressioni di tipo includono variabili quantificate universalmente su i tipi del linguaggio.  
Esempi. List.rev in Ocaml ha tipo 'a list -> 'a list. Dato un qualunque tipo t, una qualunque espressione e di tipo t list. Allora, List.rev è applicabile e l'espressione (List.rev e) ha tipo t list.

**Ordinamento parziale** permette sempre di individuare il **tipo più generale** nell'insieme di quelli associabili ad ogni struttura del programma.

**Esempi.** Seq(t), <t> Seq(t) QSort(Seq(t)S, t x t -> bool L) nel QuickSort in Java;

Una sola <t>Swap(t x, t y) su un generico t invece che tante Swap per diversi tipi utilizzati nel programma.

**Meccanismo fondamentale per riuso di codice** (presente in tutti i linguaggi di ultima generazione)

- **Polimorfismo di sottotipo.** Polimorfismo in combinazione con i sottotipi.  
**Esempi.** Java prime versioni, solo Polimorfismo di sottotipo, successive versioni Polimorfismo generico e di sottotipo (lo useremo).

# Nuovi Tipi di Dato: Una Metodologia di Definizione

Nuovi tipi di dato sono introdotti con i meccanismi delle espressioni di tipo e del loro naming

Ad esempio:

```
typedef struct LambdaTerm * LastT; //introdotta in C per AST dei lambda termini
type store = Store of int * (loc -> mval);; (* introdotta in Ocaml per lo Store della AM di Small21 *)
```

Leggendo la definizione di LastT e/o quella di store, nascono spontanee 2 fondamentali domande:

**Q:** Come appare un valore del nuovo tipo definito?

Ad esempio, il valore 5 del tipo primitivo int appare come 5. Ed invece, uno store?

**Q:** La definizione data fornisce una definizione "sensata"?

Ad esempio, il valore 5 so che è implementato con una grandezza che misura il valore 5 consistentemente a quelle utilizzate per gli altri interi. E per il tipo LastT sono sicuro che esprima anche l'AST di  $\lambda x.x$ ?

La definizione di nuovi Tipi di Dato richiede una Metodologia di Definizione che fornisca:

- **Presentazione.** Come si presentano concretamente i valori definiti (v.d.) dalla definizione di tipo (def.).
- **Adeguatezza.** Perché la def. è adeguata a supportare le operazioni (ops.) che devono equipaggiare i v.d. La definizione deve essere data pensando alle operazioni da supportare.
- **Correttezza.** Come la def. e le ops. soddisfano le proprietà attese sui v.d. Dovremmo disporre dell'insieme di proprietà che devono essere garantite nell'uso dei v.d..

# PreSAI: Una Metodologia per Nuovi Tipi di Dato

**PreSAI** è un acronimo per:

PREsentazione+Stato\_implementazione+Abstract\_function+Invariant

La metodologia, nella forma presentata, è stata ideata da Barbara Liskov e David Guttag <sup>4</sup>

La metodologia PreSAI si applica volendo introdurre, nel programma, un nuovo insieme di valori (necessari per la computazione che il programma deve fornire). Essa parte da tale esigenza e procede nei seguenti 4 passi, nell'ordine.

- **Presentazione.**  
Fornire, in opportuno formalismo, una forma per i valori di tale insieme. Indichiamo con  $V$  l'insieme di tali forme.
- **Stato.** Fornire, nel Linguaggio di Programmazione, la Definizione del Tipo di Dato, identificando lo Stato di Implementazione utilizzato per esprimere i valori definiti. Indichiamo con  $C$  l'insieme di tali stati, detti anche stati concreti.
- **Funzione di Astrazione AF.**  
Fornire, in un formalismo appropriato, una Definizione della funzione *suriettiva e parziale*,  $AF: C \rightarrow V$ .
- **Invariante di Rappresentazione I.**  
Fornire, in un formalismo appropriato, una Definizione del predicato  $I$  che identifica il sotto-insieme:  
 $C_I = \{c \in C \mid AF(c) \in V\}$ .

---

<sup>4</sup> descritta in forma estesa nel volume [LK] (vedi "Testi e Approfondimenti" nelle pagine del corso), ma già apparsa in loro precedenti articoli sulla definizione algebrica dei Tipi di Dato.

# PreSAI: Applichiamo al T.D. env di Small21

- **Ambiente.**  $\rho$  e le sue operazioni:
  - **bind**( $\rho$ , **ide**, **den**): (alias, [ide/den]  $\circ$   $\rho$ ) aggiunge un nuovo binding
  - **getEnv**( $\rho$ , **ide**): (alias,  $\rho$ (ide)) valore denotabile di un binding
  - **emptyEnv**(): (alias,  $0^\rho$ ) crea un'ambiente senza bindings

Applichiamo PreSAI all'implementazione dell'ambiente con un T.D. di nome env in Ocaml.

- **Presentazione.**

I valori hanno la seguente forma [id1/D1, ..., idk/Dk] per  $k \geq 0$ . Quindi:

$$V = \{ [id1/D1, \dots, idk/Dk] \mid (j \in [0..k]) idj \in [ide], Dj \in [den] \}^5$$

Utilizzeremo questa forma per mostrare i valori come stringhe di caratteri, toStringEnv, o stamparli, printEnv o envDump.

- **Stato.**

La definizione del nuovo T.D. OCaml è: `type env = Env of ide list * (ide -> dval)`. Quindi:

$$C = \{ Env(l, g) \mid l \in [ide\ list], g \in [ide \rightarrow dval] \}$$

- **Funzione di Astrazione AF.**

$AF(c) = []$  if  $c = Env(l, g)$  and (perOgni id,  $g(id) = Unbound$ )

$AF(c) = [id1/D1, \dots, idk/Dk]$  if  $c = Env(l, g)$  and ( $g(idj) = Dj$  per  $j \in [1..k]$  and  $g(x) = Unbound$ , altrimenti).

- **Invariante di Rappresentazione I.**

$I(c) = (c = Env(l, g) \Rightarrow (List.mem\ x\ l) \text{ iff } (g(x) \neq Unbound))$

---

<sup>5</sup> dove: [l|t] indica l'insieme dei valori di un T.D. t

## Tipi in C.

- **Sistema di Tipi.** Il linguaggio C non ha un Sistema di Tipi, ovvero ha:
  - **Dati e Tipi di dato:** Vari, inclusi interi, caratteri, record (i.e. struct), array ...
  - **Nomi ed Espressioni** per dichiarare i dati usati
  - **Nessun Sistema di Regole** per associare un unico tipo ad ogni struttura del programma

Nell'esempio sullo stuck di un programma, la variabile "a" è calcolata come avente 2 distinti e incomparabili tipi:

- 1) char, nell'assegnamento di 'e';
- 2) int nella somma con "x".

- **Uso dei tipi in C** persegue 3 principali scopi:
  - **Allocazione Statica** della memoria necessaria a contenere i dati attesi (compilatore)
  - **Allocazione Dinamica** della memoria necessaria a contenere i dati calcolati (esecuzione run-time)
  - **Annotazione** per evidenziare l'organizzazione concettuale del programma

## Union in C.

```
union Ide{T1 ide1; ...; Tn iden; };  
union Ide{T1 ide1; ...; Tn iden; } x;  
typedef union Ide{T1 ide1; ...; Tn iden; } T;
```

- **union** Ide

definisce un tipo di dato unione, visto come una struttura di memoria allocabile staticamente o dinamicamente

- **Condivisa:** da valori di  $n$  differenti tipi di dato  $T_1 \dots T_n$   
+ Ad ogni istante la struttura può contenere uno solo di tali valori
- **n-Accessi** differenti: uno per ciascun selettore  $ide_1 \dots ide_n$   
+ Il selettore usato stabilisce la struttura con cui si accede al contenuto

### Esempio.

Tipi per costante, variabile, array, record, formano in C un unico tipo di dato A se A è un union ed il valore di ciascun di tali tipi di dato è selezionabile con uno specifico selettore di A

## Union in C.

```
union Ide{T1 ide1; ...; Tn iden;};  
union Ide{T1 ide1; ...; Tn iden; } x;  
typedef union Ide{T1 ide1; ...; Tn iden; } T;
```

- **Allocazione** di valori del tipo di dato **union** Ide{T<sub>1</sub> ide<sub>1</sub>; ...; T<sub>n</sub> ide<sub>n</sub>}
- **Allocazione Statica** eseguita a compile-time su Memoria Statica  
+ via dichiarazioni di variabili (val. modificabili) per valori union:  
**union** Ide x, ..., z;
- **Allocazione Dinamica** eseguita a run-time su Memoria Dinamica  
+ via dichiarazioni di variabili per valori puntatori a union:  
**union** Ide \* x, ..., z;  
+ via operatori di allocazione dinamica esplicita:  
(**union** Ide\*) malloc(sizeof(**union** Ide))
- **Quando** usare allocazione dinamica?  
+ Quando il numero di valori da introdurre dipende dall'esecuzione del programma. (vedi esercizio L9.)

## Union in C.

```
union Ide{T1 ide1; ...; Tn iden;};  
union Ide{T1 ide1; ...; Tn iden; } x;  
typedef union Ide{T1 ide1; ...; Tn iden; } T;
```

- **Accesso** attraverso uno dei selettori: Quale?
  - **Selettore Corretto** ovvero,
    - + quello relativo all'ultima modifica
  - **Ultima modifica** dove e come si trova?
    - + è sufficiente una variabile per ricordare
    - + il programmatore deve provvedere alla gestione di tale variabile
- **Una metodologia per la gestione in C.** Usare sempre uno struct che
  - + associ ogni tipo **union** con una variabile per controllare il corretto accesso dei componenti
  - + Esempio: Nel caso di **union Ide{T<sub>1</sub> ide<sub>1</sub>; ...; T<sub>n</sub> ide<sub>n</sub>;}**

```
typedef enum{ide1, ..., iden; } Tflag;  
struct IdeElem {Tflag flag; union Ide acc;};
```

# Tipi Algebrici o Concreti

## Definition (Tipi di Dato Algebrico)

I Tipi di Dato Algebrico sono collezioni di valori (anche strutturati), detti "termini", definiti per iniezione a partire da un insieme finito  $G = \{g_i : T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G\}$  di funzioni iniettive, dette "costruttori", di nome  $g_i$ , e segnatura  $T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G$ , dove  $k \geq 0$  e  $T_G$  sia il tipo dei valori definiti con  $G$ . Allora l'insieme di tali termini è

$$\mathcal{D}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{D}^{T_{i_j}}\},$$

dove abbiamo indicato con  $\mathcal{D}^T$  l'insieme dei valori di tipo  $T$ .

## Proposition (Presentazione e Identità Sintattica)

**Presentazione.** *Un valore algebrico, o termine, ha sempre forma  $g(v_1, \dots, v_k)$ , dove  $g$  sia un costruttore di arità  $k \geq 0$  e  $v_1, \dots, v_k$  valori di tipo atteso dalla segnatura di  $g$ .*

**Identità Sintattica**  $\equiv$ . *Siano  $g(v_1, \dots, v_n)$  e  $g'(v'_1, \dots, v'_m)$  due valori di uno stesso tipo di dato algebrico. Allora,*

$$g(v_1, \dots, v_n) = g'(v'_1, \dots, v'_m) \quad \text{sse } g \equiv g' \wedge n = m \wedge v_1 = v'_1 \wedge \dots \wedge v_n = v'_m$$

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

```
type TG = g1 of Texp1,1 * ... * Texp1,k1
      | ...
      | gn of Texpn,1 * ... * Texpn,kn
```

Dove: - ...

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

```
type TG = g1 of Texp1,1 * ... * Texp1,k1
      | ...
      | gn of Texpn,1 * ... * Texpn,kn
```

- **Dove:**

- $T_G$  è il nome del tipo e deve iniziare con un carattere **minuscolo**
- $g_1, \dots, g_n$  sono i nomi dei costruttori e hanno primo carattere **maiuscolo**
- $\text{of } Texp_{i,1} * \dots * Texp_{i,k_i}$  è omissso quando  $k_i = 0$ , i.e.  $g_i$  ha arità 0
- $\text{of } Texp_{i,1} * \dots * Texp_{i,k_i}$  definisce la tupla dei tipi degli argomenti
- $Texp_{i,j}$  è una qualunque espressione di tipo Ocaml
- l'ordine dei costruttori è inessenziale
- Nel caso di tipi algebrici polimorfi:
  - $T_G$  ha prefisso la lista  $'a_1, \dots, 'a_r$  delle variabili generiche
  - $Texp_{i,j}$  può contenere variabili in  $\{'a_1, \dots, 'a_r\}$  come libere

- **Ricorsiva**  $T_G$  può occorrere come tipo in  $Texp_{i,k_i}$

- **Allocazione Dinamica** Ogni valutazione di  $g_i(v_{i,1}, \dots, v_{i,k_i})$  alloca dinamicamente una struttura di costruttore  $g_i$  avente  $k_i$  componenti di valore  $v_{i,1}, \dots, v_{i,k_i}$  in tale ordine.

# Programmazione con Tipi Algebrici o Concreti

- Programmare con Tipi Algebrici richiede non solo saper:
  - **Definire tipi algebrici.**  
Esempio, alberi ('a, 'b)gTree in OCaml  
`type('a, 'b)gTree = E | L of 'a | R of ('b * ('a, 'b)gTree list)`
  - **Esprimere valori** di un tipo di dato algebrico.  
Esempio, l'albero tree:  
`let tree = R("root1", [L 1; L 2; R("root2", [])]; E]`
- anche saper:
  - **Visitare la Struttura**  
Esempio: l'albero tree è una foglia? Ha più di 1 sottoalbero?
  - **Accedere Componenti.**  
Esempio: se tree ha sottoalberi, qual'è il suo primo sottoalbero?
  - **Modificare Componenti**  
Esempio: Ocaml permette tipi algebrici a componenti modificabili
- Operazioni per Visita, Accesso e Modifica (quando ammesso) ottenute:
  - Meccanismo di Pattern-Matching (Ocaml, e Ling. funzionali)
  - Meccanismi ad hoc (ad es.: funzioni definite per casi)

# Pattern-Matching

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- È realizzato mediante:
  - **Patterns** per tipi di dato algebrico
  - Operazione **Match** per coppie pattern-termini
- Sia  $G = \{g_i : T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G\}$  l'insieme dei costruttori di un Tipo Algebrico  $T_G$ , e indichiamo con  $X^T$  l'insieme delle variabili di tipo  $T$ 
  - **Termini**.  $\mathcal{D}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{D}^{T_{i_j}}\}$   
Ad esempio:  $R(\text{"root1"}, [L\ 1; L\ 2; R(\text{"root2"}, []); E])$  è un termine di tipo  $(\text{'a'}, \text{'b'})\text{gTree}$ , in Ocaml.
  - **Patterns**  $\mathcal{P}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{P}^{T_{i_j}}\} \cup X^{T_G}$   
Ad esempio<sup>6</sup>:  $R(x, [L\ w; L\ 2; y; z])$  è un pattern dove:  $x \in X^{T_1}$ ,  $w \in X^{T_2}$ ,  $y, z \in X^{T_3}$  sono variabili libere<sup>7</sup> di tipo  $T_1 = \text{string}$ ,  $T_2 = \text{int}$ ,  $T_3 = (\text{int}, \text{string})\text{gTree}$ .
- Operazione Match: Dato  $T_G$ ,
  - **Match** :  $\mathcal{D}^{T_G} \times \mathcal{P}^{T_G} \rightarrow \text{Sostituzione}$

<sup>6</sup>Quando  $T$  non è Algebrico,  $\mathcal{P}^T \equiv \mathcal{D}^T \cup X^T$ . Vedi esercizio L9.12.

<sup>7</sup>ovvero, non introdotte da alcun binder nel termine in cui il pattern occorre

# Pattern-Matching in Ocaml

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- Operazione Match: Dato  $T_G$ ,
  - $\text{Match} : \mathcal{D}^{T_G} \times \mathcal{P}^{T_G} \rightarrow \text{Sostituzione}$
  - Sostituzione è l'insieme dei legami di variabili-valori che rendono il pattern uguale al termine dato, oppure *fail* se tale insieme non esiste.

Esempio:

$\text{Match}(\text{R}(\text{"root1"}, [\text{L } 1; \text{L } 2; \text{R}(\text{"root2"}, [])]; \text{E}), \text{R}(x, [\text{L } w; \text{L } 2; y; z]))$   
calcola la sostituzione  $\{x = \text{"root1"}, w = 1, y = \text{R}(\text{"root2"}, []), z = \text{E}\}$

- Ocaml incorpora l'operazione Match in un costrutto `match_with`.
- Costrutto `match_with` è un condizionale multi-way avente la seguente forma:

```
match term with
  | pattern1 -> exp1
  | ...
  | patternn -> expn
```

**dove** `term` è un valore di un tipo algebrico  $T$ , e `pattern1, ..., patternn` sono  $n$  patterns per lo stesso tipo  $T$  che devono fornire una copertura per  $T$  (ovvero, per ogni termine di  $T$  deve esistere un pattern nella copertura e una sostituzione per esso che rende il pattern identico al termine)

**calcola** `expi` tale che  $i$  è il più piccolo indice per cui:

$\text{Match}(\text{term}, \text{pattern}_i) \neq \text{fail}$

# Pattern-Matching in Ocaml: Esempi

Pattern-Matching è un meccanismo per *visitare*, *accedere*, "*modificare*" valori e componenti di tipi algebrici o concreti

- Costrutto `match with` è un condizionale multi-way avente la seguente forma:

```
match term with
  | pattern1 -> exp1
  | ...
  | patternn -> expn
```

**dove** `term` è un valore di un tipo algebrico `T`, e `pattern1, ..., patternn` sono ...  
**calcola** `expi` tale che `i` è il più piccolo indice per cui:

`Match(term, patteri) ≠ fail`

## Esempio (Calcolo della sostituzione)

*Definiamo un tipo per la presentazione delle sostituzioni:*

```
type 'a subst = Sub of 'a | Fail;;
```

*ed usiamolo nel match di termini di tipo ('a, 'b)gTree definito prima.*

```
let tree = R("root1", [L 1; L 2; R("root2", []); E]);;
```

```
match tree with R(x, [L 1; L 2; y; z]) -> Sub(x, y, z) | w -> Fail;;
```

*Cosa otteniamo in Ocaml?*

## Esempio (Predicati e Selettori di gTree)

*Definiamo i predicati isE, isL, isR per riconoscere i valori delle 3 diverse strutture :*

```
let isE tree = match tree with E -> true | _ -> false;;
```

```
let isL tree = match tree with L _ -> true | _ -> false;;
```

```
let isR tree = match tree with R _ -> true | _ -> false;;
```

## Problema.

Si consideri la seguente definizione C di un tipo di dato T ottenuto dall'unione dei tipi introdotta dal tipo TUnion e con accesso controllato dal tipo Tflag.

```
typedef enum{ide1, ..., iden; } Tflag;  
typedef union Sub{T1 ide1; ...; Tn iden; } TUnion;  
typedef struct Elem {Tflag flag; TUnion acc; } T;
```

Assunta la corrispondenza  $\simeq$  tra i tipi  $\{T_1, \dots, T_n\}$  di C e i tipi  $\{T_1^0, \dots, T_n^0\}$  di OCaml:

$$T_1 \simeq T_1^0, \dots, T_n \simeq T_n^0$$

in accordo alla quale consideriamo equivalenti nei rispettivi linguaggi, i valori espressi nei corrispondenti tipi.

Si esprima con un Tipo Algebrico OCaml, il Tipo Union T di C.

## Soluzione.

$$\text{type } T^0 = \text{Ide}_1^0 \text{ of } T_1^0 \mid \dots \mid \text{Ide}_n^0 \text{ of } T_n^0$$

Ed abbiamo:

- $T \simeq T^0$
- Espressività T.D. OCaml maggiore di T.D. C, poichè la trasformazione è da C in Ocaml (a meno di fornire una trasformazione da T.D. Ocaml in T.D. C)

- 1 (a) Si faccia un esempio di valore calcolabile ma non memorizzabile: Si dica quale linguaggio lo usa e come tale valore è introdotto e può essere usato nei programmi di tale linguaggio.  
(b) In C abbiamo valori con tali caratteristiche?
- 2 Quali sono le caratteristiche distintive dei tipi di dato?
- 3 Indicare i 4 principali scopi per la presenza dei tipi nei Linguaggi di Programmazione.
- 4 (a) Cos'è la Type Safety?  
(b) Cos'è uno stato di stuck?  
(c) Come può essere verificata la safety di un programma?
- 5 (a) In cosa equivalenza nominale e strutturale si differenziano?  
(b) In cosa il cast si differenzia dalla coercion?
- 6 (a) Cos'è una variabile di tipo e in quale meccanismo è utilizzata? (b) In cosa si differenzia l'overloading dal polimorfismo parametrico?
- 7 Si consideri la seguente definizione OCaml:  
$$\text{unC } g = \text{fun } (x, y) \rightarrow g \ x \ y$$
  
(a) Si dica quale tipo ha?  
(b) Si dica quale funzione definisce.
- 8 Il termine  $\lambda x.x \ x$  è un termine sintatticamente corretto del Lambda Calcolo e  
(a) può essere "trascritto" in OCaml ottenendo il termine ...  
(b) che per quanto sintatticamente corretto non ha un tipo associabile: Perché?