Dati, Tipi di Dato e Tipi di un Linguaggio

Sommario: 29 aprile, 2020

- 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 costrutto Union
- Studio di Casi: Tipi algebrici (o concreti) in OCaml
- Espressività
- Esercizi

Dati, Tipi di Dato, Tipi: Dato

- **Dato**: La più semplice struttura **per introdurre valori** in un programma
  - **Caratteristiche Distintive** sono i modi con cui possono essere usato nei programmi
    - Valori calcolabili: Espressi da espressione non atomica (i.e. variabile o literal);
    - Denotable values: Denotazioni in ambiente (in generale, dichiarazioni);
    - Storable values: (in genere) Valori di un modificabile;
    - Literal values: Espressi con literals.
  - 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 C */
A = {2, 7, 12, 9};/* errore in C */
```

- `{2,7,12,9}` esprime in C, un valore array solo nella dichiarazione
- una funzione in C 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`
    - **Memorizzatili, 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[1]) 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"

[1] ma per i principi della Programmazione Strutturata, ciò si ripete ad ogni livello con i sotto-problemi

Dati, Tipi di Dato, Tipi: Tipi/2

- 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[2].
    Esempi. È il caso del seguente programma C.

[2] 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

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

```c
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)

[3] 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. $\mathtt{T = Void + Int + Char + ...}$
  - (Linguaggio delle) **Espressioni di Tipo** con cui possono essere definiti tipi, anche derivati
    Esempi. $\mathtt{T = ...+\ T \times ... \times T \rightarrow T\ + ...}$
  - **Regole (di inferenza)** con cui il sistema associa un unico tipo a ogni struttura del programma
    Esempi.

$$\frac{\mathtt{E : bool} \quad \mathtt{C : Void}}{\mathtt{while\ E\ do\ C : Void}} \qquad \frac{\mathtt{E : bool} \quad \mathtt{E_{then} : T1} \quad \mathtt{E_{else} : T2} \quad \mathtt{T1 = T = T2}}{\mathtt{E\ ?\ E_{then}\ E_{else} : T}}$$

- 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 Small20 è un esempio di sistema (da provare) sound per il Linguaggio Small20.

Relazioni e Proprietà sulla struttura dei Tipi

Intervengono quando **non c'è identità** tra tipo atteso e tipo effettivo.
Esempio. {.. `T1 x;` ... `x = E;` .. }. Il tipo atteso per `E` è `T1`, ma `E` ha tipo `T2`≠`T1`.

- **Equivalenza**. Sia `x:T` and `y:T'`. Quando `x` e `y` hanno stesso tipo?
  - **Nominal**. Ogni tipo è uguale solo a sè stesso.
    Esempi. {**type** `T1 = int; T1 x; int y; x=y`} contiene un errore di tipo
  - **Structural**. Stessa espressione di Tipo quando i nomi sono rimpiazzati dalle definizioni.
    Esempi. {**type** `T1 = int; T1 x; int y; x=y`} sequenza corretta
- **Compatibilità e Sottotipo**
  - `T` è compatibile con `S` se (un valore di tipo) `T` può essere usato quando è atteso `S`.
    Esempi. Sottotipi `T`⊂`S` in Java:
    type $\mathtt{S} = \mathtt{d_S}$; ...**type** $\mathtt{T} \subset \mathtt{S} = \mathtt{d_T}$; ....$\mathtt{T}\,\mathtt{x} = \mathtt{e_x}$; ....$\mathtt{S}\,\mathtt{y} = \mathtt{x}$;
- **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** $\mathtt{T} \subset \mathtt{S} = \mathtt{d_T}$; ....$\mathtt{T}\,\mathtt{x} = \mathtt{e_x}$; ....$\mathtt{S}\,\mathtt{y} = \mathtt{x}$; ....; $\mathtt{T}\,\mathtt{z} = (\mathtt{T})\mathtt{y}$;

Relazioni e Proprietà sulla struttura dei Tipi /2

Intervengono quando **non c'è identità** tra tipo atteso e tipo effettivo.
Esempio. {`T1 x; ...; 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.

  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`$\times$`t` $\rightarrow$ `bool L)` nel QuickSort;
  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**. Polimorfimo in combinazione con i sottotipi.
  Esempi. Java prime versioni, solo Polimorfimo 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; //introdotto in C per AST dei lambda termini
type store = Store of int * (loc -> mval);; (* introdotto in Ocaml per lo Store della AM di Small20 *)
```

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.** Perchè la def. è adeguata a supportare le operazioni che devono equipaggiare (ops.) 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:
PREntazione+Stato_implementazione+Abstract_function+Invariant

La metodologia, nella forma presentata, è stata ideata da Barbara Liskov e David Guttag [4]

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 -> V`.
- **Invariante di Rappresentazione I.**
  Fornire, in un formalismo appropriato, una Definizione del predicato I che identifica il sotto-insieme:
  $\mathtt{C_I} = \{\mathtt{c} \in \mathtt{C} \mid \mathtt{AF(c)} \in \mathtt{V}\}$.

[4] 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 Small20

- **Ambiente**. ρ e le sue operazioni:
  - **bind**(ρ, **ide**, **den**): (alias, [ide/den]$\circ\rho$) aggiunge un nuovo binding
  - **getEnv**(ρ, **ide**): (alias, ρ(ide)) valore denotabile di un binding
  - **emptyEnv**(): (alias, 0^{ρ}) 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>=0`. Quindi:
  $\texttt{V} = \{[\texttt{id1/D1}, .., \texttt{idk/Dk}] \mid (\texttt{i} \in [0..\texttt{k}])\texttt{idi} \in [|\texttt{ide}|], \texttt{Di} \in [|\texttt{den}|]\}$[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:
  $\texttt{C} = \{\texttt{Env(l, g)} \mid \texttt{l} \in [|\texttt{ide list}|], \texttt{g} \in [|\texttt{ide -> dval}|]\}$

- **Funzione di Astrazione AF.**
  ```
  AF(c) = [] if c = Env(l,g) and (perOgni id, g(id) = Unbound)
  AF(c) = [id1/D1,...,idk/Dk] if c = Env(l,g) and (g(idi) = Di per i∈[1..k] and
                                                   g(x) = Unbound altrimenti).
  ```

- **Invariante di Rappresentazione I.**
  ```
  I(c) = (c = Env(l,g) => (List.mem x l) iff (g(x) != Unbound)
  ```

[5] dove: `[|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

Costrutto Union in C: Struttura e Comportamento

Union in C.

union Ide{T_1 ide_1; ...; T_n ide_n;};
union Ide{T_1 ide_1; ...; T_n ide_n;} x;
typedef union Ide{T_1 ide_1; ...; T_n ide_n;} 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 ... T_n
    + Ad ogni istante la struttura può contenere uno solo di tali valori

  - **n-Accessi** differenti: uno per ciascun selettore ide_1 ... 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

Costrutto Union in C: Allocazione di Memoria

Union in C.

union Ide{T_1 ide_1; ...; T_n ide_n;};
union Ide{T_1 ide_1; ...; T_n ide_n;} x;
typedef union Ide{T_1 ide_1; ...; T_n ide_n;} T;

- **Allocazione** di valori del tipo di dato **union** Ide{T_1 ide_1; ...; T_n ide_n}
  - **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.)

Costrutto Union in C: Accedere per Leggere/Modificare

Union in C.

union Ide{T_1 ide_1; ...; T_n ide_n;};
union Ide{T_1 ide_1; ...; T_n ide_n;} x;
typedef union Ide{T_1 ide_1; ...; T_n ide_n;} 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_1 ide_1; ...; T_n ide_n;}

    typedef enum{ide_1, ..., ide_n;} 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 $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ di funzioni iniettive, dette "costruttori", di nome $\mathtt{g_i}$, e segnatura $\mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}$, dove $\mathtt{k} \geq 0$ e $\mathtt{T_G}$ sia il tipo dei valori definiti con $\mathtt{G}$. Allora l'insieme di tali termini è

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

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

Proposition (Presentazione e Identità Sintattica)

Presentazione. *Un valore algebrico, o termine, ha sempre forma* $\mathtt{g(v1, ..., vk)}$, *dove* $\mathtt{g}$ *sia un costruttore di arità* $\mathtt{k} \geq 0$ *e* $\mathtt{v1, \ldots, vk}$ *valori di tipo atteso dalla segnatura di* $\mathtt{g}$.
Identità Sintattica $\equiv$. *Siano* $\mathtt{g(v1, ..., vn)}$ *e* $\mathtt{g'(v1', ..., vm')}$ *due valori di uno stesso tipo di dato algebrico. Allora,*

$$\mathtt{g(v1, ..., vn)} = \mathtt{g'(v1', ..., vm')} \quad \textit{sse } \mathtt{g} \equiv \mathtt{g'} \wedge \mathtt{n} = \mathtt{m} \wedge \mathtt{v1} = \mathtt{v1'} \wedge ... \wedge \mathtt{vn} = \mathtt{vm'}$$

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

$$\begin{aligned} \mathtt{type\ T_G} = {} & \mathtt{g_1\ of\ Texp_{1,1} \star ... \star Texp_{1,k_1}} \\ & \mid ... \\ & \mid \mathtt{g_n\ of\ Texp_{n,1} \star ... \star Texpt_{n,k_n}} \end{aligned}$$

Dove: - ...

Ocaml: Tipi Algebrici o Concreti

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

$$\begin{aligned}\texttt{type } \mathtt{T_G} = \mathtt{g_1} \texttt{ of } \mathtt{Texp_{1,1}} \star \ldots \star \mathtt{Texp_{1,k_1}} \\ \mid \ldots \\ \mid \mathtt{g_n} \texttt{ of } \mathtt{Texp_{n,1}} \star \ldots \star \mathtt{Texpt_{n,k_n}}\end{aligned}$$

  - **Dove:**
    - $\mathtt{T_G}$ *è il nome del tipo e deve iniziare con un carattere* **minuscolo**
    - $\mathtt{g_1}, .., \mathtt{g_n}$ *sono i nomi dei costruttori e hanno primo carattere* **maiuscolo**
    - of $\mathtt{Texp_{i,1}} \star \ldots \star \mathtt{Texp_{i,k_i}}$ *è omesso quando* $\mathtt{k_i} = 0$, *i.e.* $\mathtt{g_i}$ *ha arità 0*
    - of $\mathtt{Texp_{i,1}} \star \ldots \star \mathtt{Texp_{i,k_i}}$ *definisce la tupla dei tipi degli argomenti*
    - $\mathtt{Texp_{i,j}}$ è una qualunque espressione di tipo Ocaml
    - l'ordine dei costruttori è inessenziale
    - Nel caso di tipi algebrici polimorfi:
      - $\mathtt{T_G}$ ha prefisso la lista $\texttt{'a}_1, \ldots, \texttt{'a}_r$ delle variabili generiche
      - $\mathtt{Texp_{i,j}}$ può contenere variabili in $\{\texttt{'a}_1, \ldots, \texttt{'a}_r\}$ come libere

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

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

Programmare 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-termine

- Sia $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ l'insieme dei costruttori di un Tipo Algebrico $\mathtt{T_G}$, e indichiamo con $\mathtt{X^T}$ l'insieme delle variabili di tipo `T`
  - **Termini**. $\mathcal{D}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{D}^{\mathtt{T_{i_j}}}\}$
    Ad esempio: `R("root1",[L 1;L 2;R("root2",[]);E])` è un termine di tipo `('a,'b)gTree`, in Ocaml.
  - **Patterns** $\mathcal{P}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{P}^{\mathtt{T_{i_j}}}\} \cup \mathtt{X^{T_G}}$
    Ad esempio[6]: $\mathtt{R(x,[L\ w;L\ 2;y;z])}$ è un pattern dove: $\mathtt{x} \in \mathtt{X^{T1}}$, $\mathtt{w} \in \mathtt{X^{T2}}$, $\mathtt{y}, \mathtt{z} \in \mathtt{X^{T3}}$ sono variabili libere[7] di tipo `T1 = string`, `T2 = int`, `t3 = (int,string)gTree`.

- Operazione Match: Dato $\mathtt{T_G}$,
  - $\mathtt{Match} : \mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow \mathtt{Sostituzione}$
  - ....

[6] Quando `T` non è Algebrico, $\mathcal{P}^{\mathtt{T}} \equiv \mathcal{D}^{\mathtt{T}} \cup \mathtt{X^T}$. Vedi esercizio L9.12.

[7] ovvero, non introdotte da alcun binder nel termine in cui il patter 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 $\mathtt{T_G}$,
  - $\mathtt{Match} : \mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow \mathtt{Sostituzione}$
  - 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:
    $\mathtt{Match}(\mathtt{R}(\text{"root1"}, [\mathtt{L\ 1}; \mathtt{L\ 2}; \mathtt{R}(\text{"root2"}, []); \mathtt{E}]), \mathtt{R}(\mathtt{x}, [\mathtt{L\ w}; \mathtt{L\ 2}; \mathtt{y}; \mathtt{z}]))$
    calcola la sostituzione $\{\mathtt{x} = \text{"root1"}, \mathtt{w} = 1, \mathtt{y} = \mathtt{R}(\text{"root2"}, []), \mathtt{z} = \mathtt{E}\}$

- 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
      | pattern_1 -> exp_1
      | ...
      | pattern_n -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e $\mathtt{pattern}_1, \ldots, \mathtt{pattern}_n$ 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 $\mathtt{exp}_i$ tale che `i` è il più piccolo indice per cui:

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

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
      | pattern_1 -> exp_1
      | ...
      | pattern_n -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e $\texttt{pattern}_1, \ldots, \texttt{pattern}_n$ sono ...
calcola $\texttt{exp}_i$ tale che `i` è il più piccolo indice per cui:

$$\texttt{Match}(\texttt{term}, \texttt{patter}_i) \neq \texttt{fail}$$

Esempio (Calcolo della sostituzione)

Definiamo un tipo per la presentazione delle sostituzioni:

```
type 'a substs = 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;;
```

Espressività

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$\{\texttt{ide}_1, ..., \texttt{ide}_n;\}$ Tflag;
typedef union Sub$\{\texttt{T}_1\ \texttt{ide}_1; ...; \texttt{T}_n\ \texttt{ide}_n;\}$ TUnion;
typedef struct Elem $\{$Tflag flag; TUnion acc; $\}$ T;

Assunta la corrispondenza $\simeq$ tra i tipi $\{\texttt{T}_1, ..., \texttt{T}_n\}$ di C e i tipi $\{\texttt{T}_1^0, ..., \texttt{T}_n^0\}$ di OCaml:
$\texttt{T}_1 \simeq \texttt{T}_1^0, ..., \texttt{T}_n \simeq \texttt{T}_n^0$
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.

type $\texttt{T}^0 \ = \ \texttt{Ide}_1^0$ of $\texttt{T}_1^0 \mid ... \mid \texttt{Ide}_n^0$ of $\texttt{T}_n^0$

Ed abbiamo:

- $\texttt{T} \simeq \texttt{T}^0$
- 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)

Esercizi

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:
   $$\texttt{unC g} = \texttt{fun (x, y)} \rightarrow \texttt{g x y}$$
   (a) Si dica quale tipo ha?
   (b) Si dica quale funzione definisce.
8. Il termine $\lambda \texttt{x.x x}$ 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: Perchè?

Altri esercizi in EserciziL9.pdf