



Prima degli oggetti:

Record e polimorfismo
di sottotipo

Record (Struct)

- Introdotti per manipolare in modo unitario dati di tipo eterogeneo
- Li troviamo in C, C++, OCaml,
- Javascript: non ha tipi record, riassunti dagli oggetti
- Record possono essere annidati

Esempio (C)

```
struct studente {  
    char nome[20];  
    int matricola;  
};
```

```
studente s;  
s.matricola = 343536;
```

Esempio JavaScript (aka Object)

```
const CartoonCharacter =  
{  
  Name: "Goofy",  
  Created: 1932,  
  FirstAppeared: "Mickey's Revue"  
};
```

Record: implementazione nel compilatore

Memorizzazione sequenziale dei campi. Possibili approcci:

- Allineamento dei campi alla parola di memoria (32/64 bit): un campo per ogni parola
 - spreco di memoria (per i tipi di dato "piccoli": char, short, byte, ...)
 - accesso semplice
- Memorizzazione come packet record
 - disallineamento
 - accesso più costoso

Record: implementazione con allineamento alla parola (es. 32 bit)

```
struct x
{
    char a;        // 1 byte
    int b;         // 4 byte
    short c;       // 2 byte
    char d;        // 1 byte
};
```

L'allineamento alla parola determina uno spreco di occupazione di memoria

Record: implementazione con allineamento alla parola (es. 32 bit)

```
// effettivo "memory layout" (C COMPILER)
struct x {
    char a;           // 1 byte
    char _pad0[3];    // padding 'b' su 4 byte
    int b;            // 4 byte
    short c;          // 2 byte
    char d;           // 1 byte
    char _pad1[1];    // padding sizeof(x)
                    // multiplo di 4
}
```

Record: implementazione con allineamento alla parola (es. 32 bit)

Non sono veri campi, ma solo spazio aggiunto dal compilatore per rispettare l'allineamento

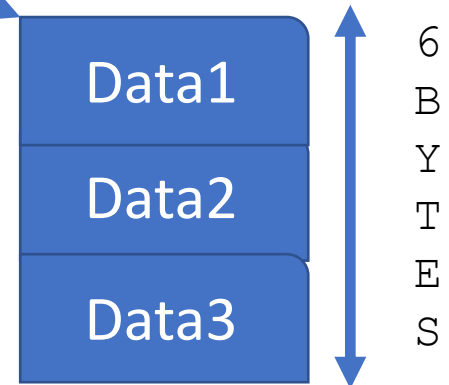
```
// effettivo "memory layout"
struct x_{
    char a;           // 1 byte
    char _pad0[3];    // padding 'a' su 4 byte
    int b;            // 4 byte
    short c;          // 2 byte
    char d;           // 1 byte
    char _pad1[1];    // padding sizeof(x_)
                      // multiplo di 4
}
```


Record: la strategia dei packet record

Indirizzo di base

- **struct MyData { short Data1;
 short Data2;
 short Data3; };**

- Assunzione: "short" = 2 byte di memoria.



- Implementazione sequenziale efficiente (in spazio): 2-byte aligned.
 - Data1 -- offset 0,
 - Data2 -- offset 2,
 - Data3 -- offset 4.
- Complessivamente bastano **6 byte** al posto dei 12 richiesti dall'allineamento alla parola (con parole di 32 bit)
- D'altro canto, la lettura di un campo richiede di leggere una parola ed estrarre il campo da esso (cosa non necessaria con l'allineamento alla parola)

Altro esempio

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

Compilazione 32-bit x86

A char (one byte) will be 1-byte aligned.

A short (two bytes) will be 2-byte aligned.

An int (four bytes) will be 4-byte aligned.

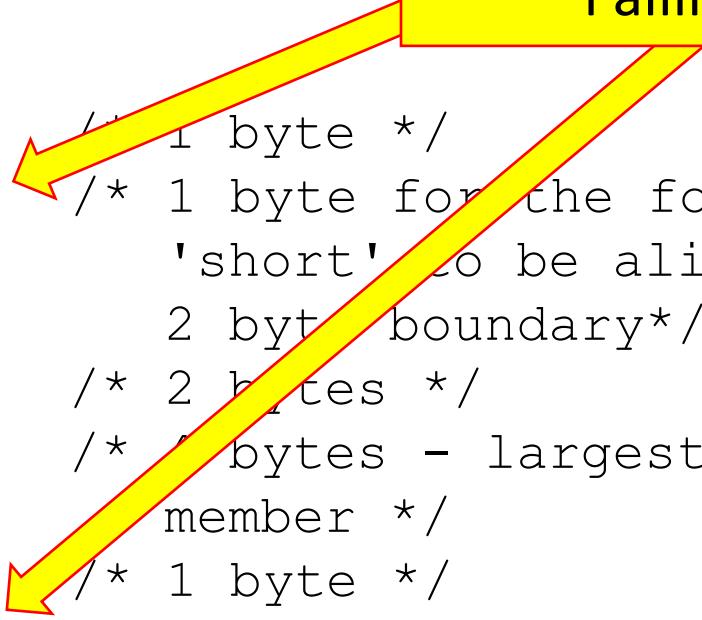
Compilazione su 32 bit x86 (con strategia packet records)

```
struct MixedData
{
    char Data1;           /* 1 byte */
    char Padding1[1];     /* 1 byte for the following
                           'short' to be aligned on a
                           2 byte boundary*/
    short Data2;          /* 2 bytes */
    int Data3;            /* 4 bytes - largest structure
                           member */
    char Data4;           /* 1 byte */
    char Padding2[3];     /* 3 bytes to make total size of
                           the structure 12 bytes */
};
```

Compilazione su 32 bit x86 (con strategia packet record)

Non sono veri campi, ma solo spazio aggiunto dal compilatore per rispettare l'allineamento

```
struct MixedData
{
    char Data1;           /* 1 byte */
    char Padding1[1];     /* 1 byte for the following
                           'short' to be aligned on a
                           2 byte boundary*/
    short Data2;          /* 2 bytes */
    int Data3;            /* 4 bytes - largest structure
                           member */
    char Data4;           /* 1 byte */
    char Padding2[3];     /* 3 bytes to make total size of
                           the structure 12 bytes */
};
```



Di nuovo spazio sprecato...

Piccola ottimizzazione (fatta dal compilatore)

```
struct MixedData /* field-reordering */  
{  
    char Data1;  
    char Data4;  
    short Data2;  
    int Data3;  
};
```

Bastano 8 byte

Record: modello formale

Sintassi espressioni

Exp ::= ...

| [$l_1: e_1, \dots l_n: e_n$]

Record

| $e.l$

Accesso

Valori

$V ::= \dots$

| [$l_1: v_1, \dots l_n: v_n$]

Record Values

Tipi

$\tau ::= \dots$

| [$l_1: \tau_1, \dots l_n: \tau_n$]

Record Types

Record: Regole di tipo (semantica statica)

$$\frac{\forall i \in [1, k] . \Gamma \vdash e_i : \tau_i}{[l_1 : e_1, \dots, l_k : e_k] \rightarrow [l_1 : \tau_1, \dots, l_k : \tau_k]}$$

$$\frac{\Gamma \vdash e : [l_1 : \tau_1, \dots, l_k : \tau_k], \quad 1 \leq j \leq k}{\Gamma \vdash e.l_j : \tau_j}$$

Record: regole di valutazione (sem. dinamica)

$$\frac{\forall i \in [1, k]. e_i \rightarrow v'_i}{[l_1: e_1, \dots, l_k: e_k] \rightarrow [l_1: v_1, \dots, l_k: v_k]}$$

$$\frac{e \rightarrow [l_1: v_1, \dots, l_k: v_k] \quad 1 \leq i \leq k}{e.l_i \rightarrow v_i}$$

record: implementazione in OCaml

```
type label = Lab of string  
type exp = ...  
    | Record of (label * exp) list  
    | Select of exp * label
```

```
Record [(Lab "size", Int 7); (Lab "weight", Int 255)]
```

Funzioni di valutazione

```
(* accesso al campo con label l *)  
let rec lookupRecord recordbody (Lab l) =  
  match recordbody with  
  | [] -> raise FieldNotFound  
  | (Lab l', v)::t ->  
    if l = l' then v else lookupRecord t (Lab l)
```

Interprete

$$\frac{\forall i \in [1, k]. e_i \rightarrow v'_i}{[l_1 : e_1, \dots, l_k : e_k] \rightarrow [l_1 : v_1, \dots, l_k : v_k]}$$
$$\frac{e \rightarrow [l_1 : v_1, \dots, l_k : v_k] \quad 1 \leq i \leq k}{e.l_i \rightarrow v_i}$$

let rec eval e = match e with

...

| Record(rbody) -> Record(evalRecord rbody)

| Select(e, l) -> match eval e with

| Record(rbody) -> lookupRecord rbody l

| _ -> raise TypeMismatch

and evalRecord recordbody =

match recordbody with

| [] -> []

| (Lab l, e)::t -> (Lab l, eval e)::evalRecord t

Record e polimorfismo

I record suggeriscono un nuovo concetto di polimorfismo:

- Ad esempio: consideriamo la seguente funzione in Lambda-calcolo tipato/MiniCaml esteso con record:

$$(\mathit{fun} \ r: [x: \mathit{Nat}] = r.x)$$

- La funzione prende un record con un campo di nome x e restituisce il valore associato al campo di nome x
- Possiamo quindi applicarla all'argomento $[x: 0]$

$$\mathit{Apply}((\mathit{fun} \ r: [x: \mathit{Nat}] = r.x), [x: 0]) \rightarrow 0$$

Record e polimorfismo

I record sono...

- Ad esempio, con record

Domanda: è possibile applicare la stessa funzione anche al record $[x: 0, y: 1]$?

- La funzione restituisce il valore a

- Possiamo quindi applicarla all'argomento $[x: 0]$

$Apply((fun r: [x: Nat] = r.x), [x: 0]) \rightarrow 0$

Record e polimorfismo

I record sono...

- Ad esempio, con record

Domanda: è possibile applicare la stessa funzione anche al record $[x: 0, y: 1]$?

Risposta: in linea di principio si (istanziando r con $[x: 0, y: 1]$ il corpo viene valutato senza problemi), ma

l'espressione non supera il type checking

- La funzione restituisce il valore a

- Possiamo quindi applicarla all'argomento $[x: 0]$

$Apply((fun\ r: [x: Nat] = r.x), [x: 0]) \rightarrow 0$

Tipi delle funzione

Ricordiamo la regola di tipo della chiamata di funzione

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \textit{Apply}(e_1, e_2) : \tau_2}$$

... e una sua istanza per l'espressione

Apply((*fun* *r*: [*x*: *Nat*] = *r.x*), [*x*: 0, *y*: 1])

Tipi delle funzione

Ricordiamo la regola di tipo della chiamata di funzione

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \textit{Apply}(e_1, e_2) : \tau_2}$$

... e una sua istanza per l'espressione

Apply((*fun* *r*: [*x*: *Nat*] = *r.x*), [*x*: 0, *y*: 1])

Nel sistema di tipo l'applicazione non è tipabile.

Il tipo dell'argomento attuale è il tipo di un record [*x*:*Nat*,*y*:*Nat*], mentre la funzione accetta un record di tipo [*x*:*Nat*].

Ma la funzione richiede solo che il suo argomento sia un record con un campo *x*; non le importa quali altri campi possa avere o meno

Tipi delle funzione

Ricordiamo la regola di tipo della chiamata di funzione

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{Apply}(e_1, e_2) : \tau_2}$$

... e una sua istanza per l'espressione

Apply((*fun* *r*: [*x*: *Nat*] = *r.x*), [*x*: 0, *y*: 1])

Nel sistema di tipo l'applicazione non è tipabile.

Il tipo dell'argomento è il tipo di un record [*x*:*Nat*,*y*:*Nat*],
mentre la funzione ha tipo *Nat* → *Nat*.

Ma la funzione **Ci servirebbe una funzione polimorfa!** che prenda in input un record con un campo *x*; non le importa quali altri campi possa avere o meno

Polimorfismo

Una funzione è polimorfa se ha la caratteristica di essere applicabile ad argomenti di tipo diverso.

OCAML

//Funzione identità polimorfa

```
# fun x -> x;;
```

```
- : 'a -> 'a = <fun>
```

```
# let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

```
# id 5;;
```

```
- : int = 5
```

```
# id "ciao";;
```

```
- : string = "ciao"
```

Polimorfismo: esempio classico

```
# let rec map (f:'a -> 'b) (xs:'a list) : 'b list =  
  match xs with  
  | [] -> []  
  | hd::tl -> (f hd)::(map f tl);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Variabili di tipo (aka "tipi parametrici"). In Ocaml possono essere inferite

```
# let rec map f xs =  
  match xs with  
  | [] -> []  
  | hd::tl -> (f hd)::(map f tl);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Polimorfismo TypeScript

```
function identity<Type>(arg: Type): Type {  
    return arg;  
}
```

```
let output = identity("myString");  
let output: string
```

Funzione polimorfa
(ancora variabili di
tipo come parametri)

Polimorfismo TypeScript

```
class GenericNumber<NumType> {  
    zeroValue: NumType;  
    add: (x: NumType, y: NumType) => NumType;  
}
```

```
let myGenericNumber = new GenericNumber<number>();  
myGenericNumber.zeroValue = 0;  
myGenericNumber.add = function (x, y) {  
    return x + y;  
};
```

Classe con Generics
(ancora parametri di
tipo...)

Polimorfismo

```
function identity<Type>(arg: Type): Type {  
  return arg;  
}
```

```
# fun x -> x;;  
- : 'a -> 'a = <fun>
```

```
# let rec map f xs =  
  match xs with  
  | [] -> []  
  | hd::tl -> (f hd)::(map f tl);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
class GenericNumber<NumType> {  
  zeroValue: NumType;  
  add: (x: NumType, y: NumType) => NumType;  
}
```

Sono tutti esempi di:
POLIMORFISMO
PARAMETRICO
(basato su tipi con parametri)

Polimorfismo per sottotipo (subsumption)

Introduciamo una nuova nozione di polimorfismo, il **polimorfismo per sottotipo**

Idea:

- si basa sull'assunzione che valori di un certo tipo **T1** possano **sempre** essere usati al posto di valori di un altro tipo **T2** (**T1** è **sottotipo** di **T2**, si scrive **T1 <: T2**)
- estende il sistema di tipi consentendo ai valori di tipo **T1** di essere considerati **anche** di tipo **T2** (regola di **subsumption**)

Polimorfismo per sottotipo

Quindi:

$$[x: \mathit{Nat}, y: \mathit{Nat}] <: [x: \mathit{Nat}]$$

poiché un record con i campi x e y può essere usato ovunque sia richiesto un record con un campo x

Polimorfismo per sottotipo

```
class OggettiDiValore {  
    name: string;  
    year: number;  
}
```

```
class Quadri {  
    name: string;  
    year: number;  
    painter: string;  
}
```

La classe Quadri è sottotipo di Oggetti di valore

Dall'intuizione alla formalizzazione

Possiamo formalizzare questa intuizione introducendo

(1) una **relazione di sottotipo**, $S <: T$, S è un sottotipo di T

(2) una regola di **subsumption** che afferma che, se $S <: T$, allora ogni valore di tipo S può anche essere considerato di tipo T

Regola di subsumption

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

Esempio: Record

$[x: \text{Nat}, y: \text{Nat}] <: [x: \text{Nat}]$

Usando la regola di subsumption

$\emptyset \vdash [x = 0, y = 1] : [x: \text{Nat}]$

L'applicazione è tipata correttamente

$\text{Apply}((\text{fun } r: [x: \text{Nat}] = r.x), [x: 0, y: 1])$

$$\frac{\frac{(r, [x: Nat]) \vdash r.x: Nat}{\emptyset \vdash (fun\ r: [x: Nat] = r.x): Nat} \quad \frac{\frac{\emptyset \vdash [x: 0, y: 1]: [x: Nat, y: Nat] \quad [x: Nat, y: Nat] <: [x: Nat]}{\emptyset \vdash [x: 0, y: 1]: [x: Nat]}}{\emptyset \vdash Apply((fun\ r: [x: Nat] = r.x), [x: 0, y: 1]): Nat}$$

Record: Relazione di sottotipo $<$:

Definizione di $<$: per i record

$$[l_1:\tau_1, \dots, l_n:\tau_n, \dots, l_{n+k}:\tau_{n+k}] <: [l_1:\tau_1, \dots, l_n:\tau_n]$$

Il tipo di un record con $n+k$ campi è un sottotipo del tipo del record n campi con stessi tipi dei singoli campi.

Record: Relazione di sottotipo $<$:

Definizione di $<$: per i record

$$[l_1:\tau_1, \dots, l_n:\tau_n, \dots, l_{n+k}:\tau_{n+k}] <: [l_1:\tau_1, \dots, l_n:\tau_n]$$

Il tipo di un record con $n+k$ campi è un sottotipo del tipo del record n campi (con stessi tipi dei singoli campi).

L'ordine dei campi è irrilevante

$$\frac{[l_1:\tau_1, \dots, l_n:\tau_n] \text{ permutazione di } [l'_1:\tau'_1, \dots, l'_n:\tau'_n]}{[l_1:\tau_1, \dots, l_n:\tau_n] <: [l'_1:\tau'_1, \dots, l'_n:\tau'_n]}$$

Record: Relazione di sottotipo

$$[l_1:\tau_1, \dots, l_n:\tau_n, \dots, l_{n+k}:\tau_{n+k}] <: [l_1:\tau_1, \dots, l_n:\tau_n]$$

$$\frac{[l_1:\tau_1, \dots, l_n:\tau_n] \text{ permutazione di } [l'_1:\tau'_1, \dots, l'_n:\tau'_n]}{[l_1:\tau_1, \dots, l_n:\tau_n] <: [l'_1:\tau'_1, \dots, l'_n:\tau'_n]}$$

Per completare la definizione:

$$\frac{\forall i. \tau_i <: \tau'_i}{[l_1:\tau_1, \dots, l_n:\tau_n] <: [l_1:\tau'_1, \dots, l_n:\tau'_n]}$$

Il sottotipo può operare in “profondità” sui campi dei record

Piccolo esercizio

Derivare, usando le regole di $<:$, la seguente relazione

$$[x: \mathbf{Nat}, y: [a: \mathbf{Nat}, b: \mathbf{Bool}], z: \mathbf{Bool}] <: [z: \mathbf{Bool}, y: [a: \mathbf{Nat}]]$$

Top e <: come preordine

È conveniente avere un tipo che sia un supertipo di ogni tipo. Introduciamo una nuova costante di tipo *Top*, più una regola che rende *Top* l'elemento massimo della relazione sottotipo.

$$S <: Top$$

Intuizione:

In java le classi sono sottotipo della classe *Object*

Inoltre, è naturale vedere <: come relazione simmetrica e transitiva:

$$S <: S$$

$$\frac{S_1 <: S_2, S_2 <: S_3}{S_1 <: S_3}$$

... e le funzioni?

Funzioni: subsumption

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Intuizione

Consideriamo una funzione f di tipo $S_1 \rightarrow S_2$, f prende in ingresso valori di tipo S_1 e potrà anche operare con argomenti di un qualsiasi sottotipo T_1 di S_1 .

Il tipo di f ci dice anche che la funzione produce come risultato valori di tipo S_2 ; questi risultati possono anche appartenere a qualunque supertipo T_2 di S_2 .

Pertanto una funzione di tipo $S_1 \rightarrow S_2$ può anche essere vista come avente tipo $T_1 \rightarrow T_2$.

Esempio

Sappiamo che:

$$[x: \mathbf{Nat}, y: \mathbf{Nat}] <: [x: \mathbf{Nat}] \quad \text{e} \quad [x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}]$$

Quindi, in accordo alla definizione, abbiamo che

$$[x: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}, y: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}]$$

che cosa significa? che una funzione che prende record che contengono x e produce record che contengono x e z può essere usata al posto di una funzione che prende record che contengono x e y e che produce record che contengono x

```
let f (g: [x: Nat, y: Nat] → [x: Nat]) (r: [x: Nat, y: Nat]) : [x: Nat] = g r ;;
```

```
let g' (r: [x: Nat]) : [x: Nat, z: Bool] = [x: r.x, y: true] ;;
```

```
f g' [x: 10, y:10] ;;
```

Esempio

Sappiamo che:

$$[x: \mathbf{Nat}, y: \mathbf{Nat}] <: [x: \mathbf{Nat}] \quad \text{e} \quad [x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}]$$

Quindi, in accordo alla definizione, abbiamo che

$$[x: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}, y: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}]$$

Ad esempio, la funzione f (in pseudo-codice simil-Ocaml) applica la funzione g al record r

che prende record che contengono x e può essere usata al posto di una funzione che prende record che contengono x e y e che produce record che contengono x

```
let f (g: [x: Nat, y: Nat] → [x: Nat]) (r: [x: Nat, y: Nat]) : [x: Nat] = g r ;;
```

```
let g' (r: [x: Nat]) : [x: Nat, z: Bool] = [x: r.x, y: true] ;;
```

```
f g' [x: 10, y: 10] ;;
```

Esempio

Sappiamo che:

$[x: \mathbf{Nat}, y: \mathbf{Nat}] <: [x: \mathbf{Nat}]$ e $[x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}]$

Quindi, in accordo alla definizione, abbiamo che

$[x: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}, z: \mathbf{Bool}] <: [x: \mathbf{Nat}, y: \mathbf{Nat}] \rightarrow [x: \mathbf{Nat}]$

Le passiamo g' il cui tipo è un sottotipo di quello di g

che prende record che contengono x e y e che produce record che contengono x e z .
pr può essere usata al posto di una
funzione che prende record che contengono x e y e che produce record che contengono x e z .

let f (r: [x: Nat, y: Nat] → [x: Nat]) (r: [x: Nat, y: Nat]) : [x: Nat] = g r ;;

let g' (r: [x: Nat]) : [x: Nat, z: Bool] = [x: r.x, y: true] ;;

f g' [x: 10, y:10] ;;

Esempio

Sappiamo che:

$[x: \text{Nat}, y: \text{Nat}] <: [x: \text{Nat}]$ e $[x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}]$

Quindi, in accordo alla definizione, abbiamo che

$[x: \text{Nat}] \rightarrow [x: \text{Nat}]$

ci si aspetta che g lavori su
record che contengono x e y.
g' non ha problemi a farlo (le
basta x)

che cosa significa? che g produce record che contengono x e y. g' non ha problemi a farlo (le basta x) posto di una funzione che prende record che contengono x e produce record che contengono x

let f (g: $[x: \text{Nat}, y: \text{Nat}] \rightarrow [x: \text{Nat}]$) (r: $[x: \text{Nat}, y: \text{Nat}]$) : $[x: \text{Nat}]$ = g r ;;

let g' (r: $[x: \text{Nat}]$) : $[x: \text{Nat}, z: \text{Bool}]$ = [x: r.x, y: true] ;;

f g' [x: 10, y:10] ;;

Esempio

Sappiamo che:

$[x: \text{Nat}, y: \text{Nat}] <: [x: \text{Nat}]$ e $[x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}]$

Quindi, in accordo alla definizione, abbiamo che

$[x: \text{Nat}] \rightarrow [x: \text{Nat}, z: \text{Bool}] <$

ci si aspetta che g produca
record che contengano x.
g' non ha problemi a farlo
(ci mette x e anche z)

che cosa significa? che una funzione g produce record che contengono x e z. g' non ha problemi a farlo (ci mette x e anche z).
funzione che prede record che contengono x e z. g' non ha problemi a farlo (ci mette x e anche z).
contengono x

let f (g: $[x: \text{Nat}, y: \text{Nat}] \rightarrow [x: \text{Nat}]$) (r: $[x: \text{Nat}, y: \text{Nat}]$) : $[x: \text{Nat}]$ = g r ;;

let g' (r: $[x: \text{Nat}]$) : $[x: \text{Nat}, z: \text{Bool}]$ = [x: r.x, y: true] ;;

f g' [x: 10, y:10] ;;

Esempio

Sappiamo che:

$[x: \text{Nat}, y: \text{Nat}] <: [x: \text{Nat}]$ e $[x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}]$

Quindi, in accordo alla definizione, abbiamo che

$[x: \text{Nat}] \rightarrow [x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}]$

che cosa significa? che una funzione g che produce record che contengono x e y è una funzione che prede record che contengono x e y e produce record che contengono x e z .
ergo, g' sarà sempre applicabile ai record passati ad f , e il risultato conterrà sempre quanto richiesto

let f ($g: [x: \text{Nat}, y: \text{Nat}] \rightarrow [x: \text{Nat}]$) ($r: [x: \text{Nat}, y: \text{Nat}]$) : $[x: \text{Nat}] = g\ r$;;

let g' ($r: [x: \text{Nat}]$) : $[x: \text{Nat}, z: \text{Bool}] = [x: r.x, y: \text{true}]$;;

$f\ g'\ [x: 10, y: 10]$;;

Esempio

Sappiamo che:

$[x: \text{Nat}, y: \text{Nat}] <: [x: \text{Nat}]$ e $[x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}]$

Quindi, in accordo alla definizione, abbiamo che

$[x: \text{Nat}] \rightarrow [x: \text{Nat}, z: \text{Bool}] <: [x: \text{Nat}, y: \text{Nat}] \rightarrow [x: \text{Nat}]$

che cosa significa? che un
produce record che contengono
funzione che prende record
contengono x

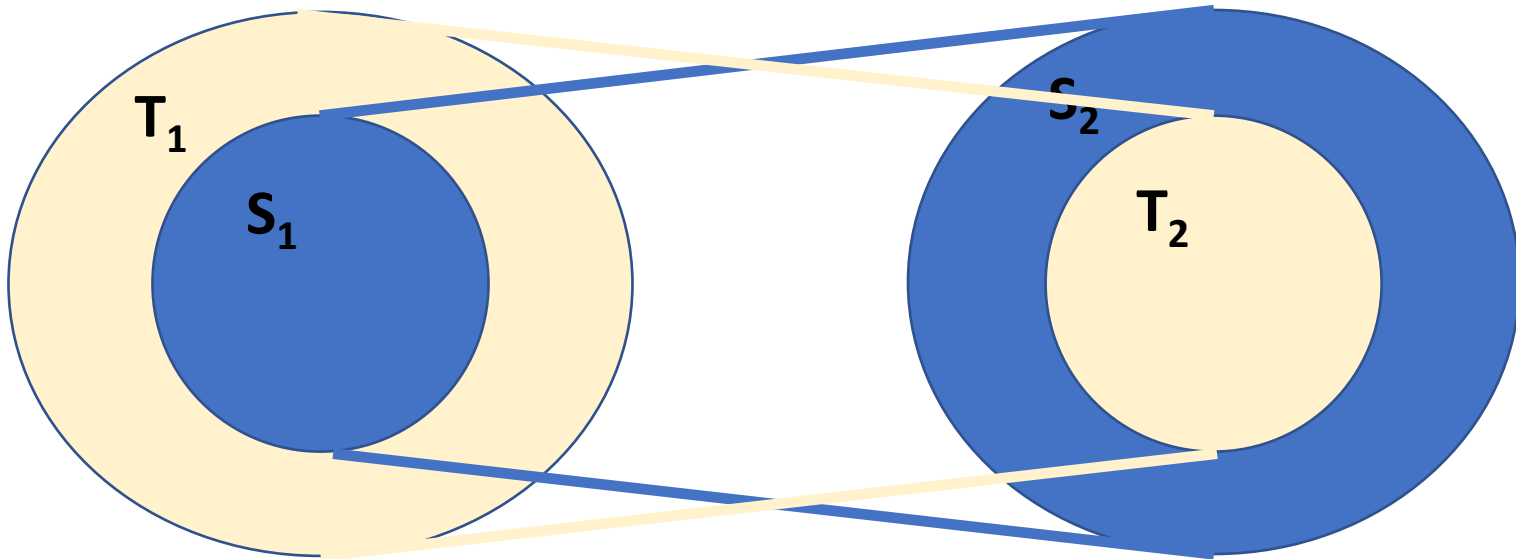
grazie alla definizione di $<:$ per le
funzioni e alla regola di subsumption,
questa applicazione è tipabile,
nonostante il primo parametro attuale
non abbia esattamente lo stesso tipo
del primo parametro formale (ma è di
un suo sottotipo)

let f (g: $[x: \text{Nat}, y: \text{Nat}]$)

let g' (r: $[x: \text{Nat}]$) = $[x: \text{Nat}, z: \text{Bool}] = [x: \text{Nat}, y: \text{Nat}]$,,

f g' [x: 10, y:10] ;;

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$



Nell'esempio di prima:

- g' si applica a tutti i record a cui si può applicare g , ma anche ad altri (a quelli che non contengono y)
- g' produce un sottoinsieme dei record che potrebbe produrre g (solo quelli che contengono anche z)

La relazione di sottotipo

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

La relazione di sottotipo è **contravariante** nella parte sinistra del tipo freccia e **covariante** nella parte destra.

Controvariante: inverte la relazione d'ordine

Covariante: rispetta l'ordine

Ma nella pratica non è sempre così:
uno sguardo a Java

I linguaggi di programmazione spesso non adottano le regole di sottotipo per record e funzioni.

In Java, una sottoclasse non può cambiare i tipi dei parametri dei metodi, ma può rendere più specifico il tipo del risultato (covariante su risultati, invariante sugli argomenti).

$$\frac{S_1 <: T_1}{m: S \rightarrow S_1 <: m: S \rightarrow T_1}$$

... ne riparleremo più avanti!

Liste: sottotipo

$$\frac{S <: T}{List\ S <: List\ T}$$

**List è un costruttore di tipo
covariante.**



Prima degli oggetti:

Tipi di dato astratti

(ADT: Abstract Data Types)

Tipi di dato astratti – Abstract Data Types

- Un **tipo di dato astratto (ADT)** consiste in un **insieme di dati** e una **collezione di operazioni per operare** sui dati di quel tipo
- **Estendibili**: Meccanismi linguistici per costruire nuovi tipi di dato astratto
- **Astratti**: La rappresentazione è nascosta agli utenti del dato
- **Incaspulati**. Si opera con i dati solo attraverso le operazioni fornite dall'ADT.
- **Distinzione tra specifica e implementazione**

ADT: specifica

La specifica di un ADT descrive il tipo di dati e le operazioni senza fornire i dettagli di implementazione: solo la semantica delle operazioni

La specifica di un ADT fornisce le informazioni necessarie per usare effettivamente il tipo nello sviluppo di programmi: Dichiarazioni della segnatura delle funzioni

ADT: Implementazione

Definizione dei dettagli di implementazione di un ADT

Implementazione non accessibile all'utente tramite delle forme di controllo dell'accesso

Implementazione dettagli su tutte le strutture di dati e tutte le operazioni

ADT: una forma di astrazione

- Astrazione per specifica: descrivono il **cosa** (**what**) delle funzionalità di un modulo di un programma e lasciando il **come** (**how**) a chi deve effettivamente realizzare la funzionalità offerta dal modulo.
- La forma di astrazione dei ADT è più chiara quando si esamina la sintassi, spesso espressa in forma algebrica.

Esempio di specifica (in pseudolinguaggio)

Stack <E>

Signature **//Operazioni significative**

new : -> STACK

push : E, STACK -> STACK

top : STACK -> E

pop : STACK -> STACK

isEmpty : STACK -> Bool

undef_e : -> E **//Completezza**

undef_s : -> STACK

Axioms **//Proprietà astratte da rispettare**

\forall e: E, stk: STACK

top(push(e, stk)) = e ;

top(new) = undef_e ;

pop(push(e, stk)) = stk ;

pop(new) = undef_s ;

isEmpty(new) ;

~isEmpty(push(e, stk))

Discussione

Abstract Data Type (ADT) è un concetto astratto definito da assiomi che rappresentano alcuni dati e operazioni su quei dati.

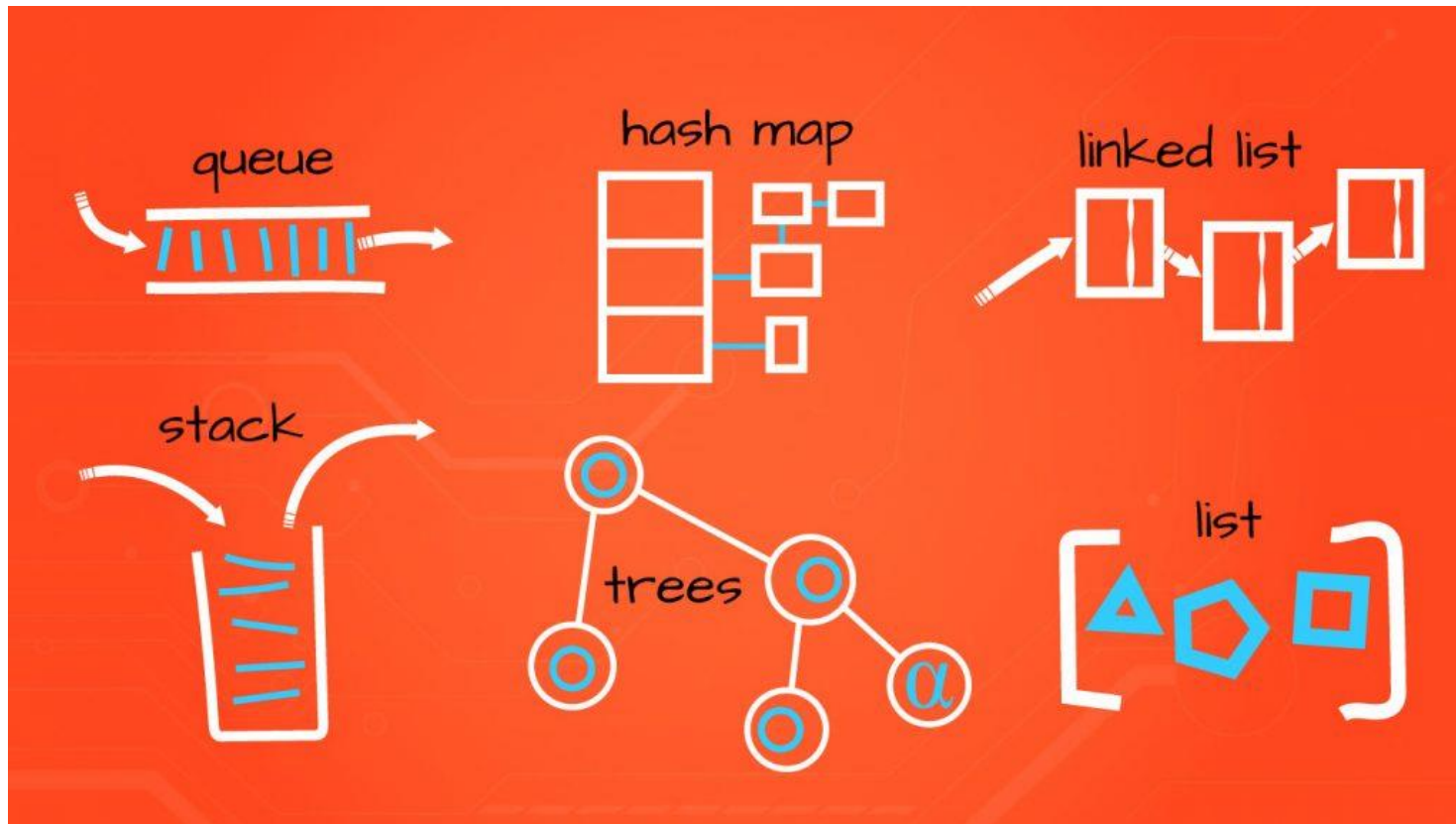
I tipi di dati astratti si concentrano su cosa, non come (sono strutturati in modo dichiarativo e non specificano algoritmi o strutture di dati).

Gli ADT ci forniscono un modo per definire formalmente i moduli riutilizzabili in un modo matematicamente valido, preciso e non ambiguo.

Gli ADT devono avere una struttura FAMED. Formale, ampiamente applicabile, minimo, estensibile e dichiarativo.

Gli ADT dovrebbero includere una descrizione leggibile dall'uomo, definizioni, firme astratte e assiomi formalmente verificabili.

ADT presenti nelle librerie standard (API) di molti linguaggi di programmazione



Un esempio: il sistema dei moduli di Ocaml

```
module type BOOL = sig
  type t
  val yes: t
  val no: t
  val choose: t -> 'a -> 'a -> 'a
```

SEGNATURA
MODULO

Introduciamo il tipo astratto **t** e lo referiamo con **BOOL**

Tipi astratti sono denotabili

Il tipo ammette due valori (**yes**, **no**) e una operazione **choose**

Segnatura dichiara la forma del tipo non l'implementazione

Una implementazione

```
module M1 : BOOL = struct
  type t = unit option
  let yes = Some ()
  let no = None
  let choose v ifyes ifno =
    match v with
    | Some () -> ifyes
    | None -> ifno
end
```

La **struttura di implementazione concreta** (rep) utilizza **option type** sul tipo **unit**

Due valori concreti per **yes** e **no** e **choose** è implementata in modo standard tramite il pattern matching

Una seconda implementazione

```
module M2 : BOOL = struct
  type t = int
  let yes = 1
  let no = 0
  let choose b ifyes ifno =
    if b = 1 then
      ifyes
    else
      ifno
end
```

La struttura di implementazione è basata su **int** con le ovvie scelte

Vedere esempio su Jupyter Notebook per maggiori dettagli

Considerazioni

Specifica: La segnatura – **BOOL** – incaspula il tipo astratto t

Implementazione: rappresentazione concreta del tipo astratto e rispettando i vincoli della segnatura su valori e operazioni

Barriere di rappresentazione

Codice del cliente del tipo non può conoscere la rappresentazione: le due implementazioni sono indistinguibili

ADT e Paradigma a Oggetti

- Le nozioni alla base dei tipi di dato astratto costituiscono gli aspetti fondazionali del paradigma a oggetti