Giorno 3 - Dati

Sommario

I dati sono classificabili in base a ciò che ci possiamo fare (associarli ad un nome, ottenerli come soluzione di una valutazione di espressione, memorizzarli), ed i tipi sono classificabili in base a chi li usa (sistema, programma). Un descrittore di dato contiene informazioni sul suo tipo, ed ha vari usi nei controlli a runtime. L'ultima parte del *giorno* consiste di una carrellata di tipi di dato, con un focus particolare sui record, ed accenni sugli array.

1 Dati, tipi, e sistemi di tipo

1.1 Classificazione

I dati possono essere:

- **Denotabili** se possono essere associati ad un nome
- **Esprimibili** se possono essere il risultato della valutazione di un'espressione complessa (diversa dal semplice nome)
- **Memorizzabili** se possono essere memorizzati e modificati in una variabile.

Esempio Le funzioni in ML sono denotabili:

```
let plus (x, y) = x+y
```

Esprimibili:

```
let plus = function(x: int) -> function(y:int) -> x + y
```

Ma **non sono memorizzabili** in una variabile.

1.1.1 Tipi di dato di sistema e di programma

I tipi di dato *di sistema* sono quelli che la macchina virtuale usa per funzionare (e.g. il run-time stack), mentre quelli *di programma* sono i tipi primitivi del linguaggio (bool, int, list...) e quelli definibili dall'utente.

1.1.2 Tipo di dato: definizione

Un tipo di dato è una collezione di valori rappresentato da

- Opportune strutture dati
- Un insieme di operazioni

Dei tipi di dato ci interessano la semantica (come si comportano) e l'implementazione.

1.1.3 Descrittori di dato

Alla rappresentazione concreta di un dato è associata un'altra struttura, detta *descrittore di dato*, che contiene una descrizione del tipo del dato. I descrittori di dato sono utili, ad esempio:

- Per fare typechecking dinamico (ossia controllare se l'uso dei tipi in un'operazione è corretto)
- Per selezionare l'operatore giusto nel caso di operazioni overloaded (e.g. + sia per somma int che float, che per concatenare...)

1.1.4 Esempi

- OCaml: controllo statico dei tipi $\implies$ i descrittori non servono.
- JavaScript: Tutto dinamico, servono i descrittori.
- Java: I descrittori contengono solo l'informazione "dinamica", perché il type checking è fatto in parte dal compilatore ed in parte dal supporto a run-time. (e.g. array: il controllo dell'accesso out-of-bound è realizzato a run-time)

Esempio strano: TypeScript fa i controlli statici, ma poi transpila su js, che fa lo stesso i controlli dinamici!

1.2 Panoramica sui tipi di dato

1.2.1 Tipi di dato base (scalar)

- Booleani:
  - **Valori**: true, false
  - **Operazioni**: or, and, not, condizionali
  - **Rappresentazione**: un byte
  - **Note**: C non ha un tipo bool.
- Char:
  - **Valori**: a, A, b, B ...
  - **Operazioni**: uguaglianza; code/decode; dipendono dal linguaggio
  - **Rappresentazione**: un byte (ASCII) o due byte (UNICODE)
- Interi:
  - **Valori**: 0, 1, -1, 2, -2 ... maxint
  - **Operazioni**: +, -, *, /...
  - **Rappresentazione**: alcuni byte (2, 4), complemento a 2;
  - **Note**: interi e interi lunghi (anche 8 byte)
  - Limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio
- Reali
  - **Valori**: Valori razionali in un certo intervallo
  - **Operazioni**: +, -, *, /...
  - **Rappresentazione**: alcuni byte (4), virgola mobile
  - **Note**: reali e reali lunghi (8 byte)
  - Problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio

1.2.2 Il tipo Void

Il tipo void (unit), ha come unico valore `()`, nessuna operazione e serve per definire il tipo di operazioni che modificano lo stato senza restituire alcun valore.

Sintassi espressioni	**Valori**
`e ::= ... | void`	`v::= ... | void`
Tipi	**Regola di tipo**
τ `::= ... | void`	$\Gamma \vdash$ `void : Void`

1.2.3 Tipi composti

1. **Record**: collezione di campi, ciascuno di un diverso tipo; un campo è selezionato col suo nome
2. **Record varianti**: Record dove solo alcuni campi (mutuamente esclusivi) sono attivi a un dato istante
3. **Array**: Funzione da un tipo indice (scalare) ad un altro tipo. Array di caratteri sono detti stringhe.
4. **Insieme**: sottoinsieme di un tipo base.
5. **Puntatore**: Riferimento ad un oggetto di un altro tipo.

1.2.4 Record (struct)

Sono stati introdotti per manipolare in modo unitario dati di altro tipo eterogeneo. Li troviamo in: C, C++, CommonLisp, Ada, Pascal, Algol68, ...
I record possono essere annidati, e sono **memorizzabili**, **esprimibili** e **denotabili**.

- Pascal non ha modo di esprimere "un valore record costante"
- C lo può fare, ma solo nell'inizializzazione (initializer)
- Uguaglianza generalmente non definita (contra: Ada)

Esempi:

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

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

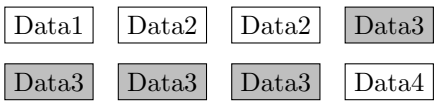
```javascript
// js (oggetto)
const CartoonCharacter =
    {Name:"Goofy",
    Created:1932,
    FirstAppeared:"Mickey's Revue"
    };
```

1.2.5 Record: implementazione

Memorizzazione sequenziale dei campi. Problema esempio:

```c
 struct MixedData
{
    char Data1;     // char: 1 byte
    short Data2;    // short: 2 byte
    int Data3;      // int: 4 byte
    char Data4;     // char: 1 byte
};
```

Se ho parole di 32 bit, e memorizzo ogni campo in base alla quantità di memoria che occupa, ho:



Data3 si trova a cavallo tra due parole!! Possibili approcci:

1. Allineamento dei campi alla parola (32/64 bit): occupo un'intera parola per ogni campo
   - Spreco di memoria (un record contiene tipi di dato diversi, che non occupano tutti lo stesso spazio in memoria)
   - Accesso semplice
2. Padding o packed record (*cerco* di mettere più valori nella stessa parola)
   - Disallineamento
   - Accesso più costoso

```c
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 */
};
```

Piccola ottimizzazione (field-reordering):

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

Tra le miriadi di opzioni del compilatore si può scegliere che modello utilizzare.

1.2.6 Record: modello

Sintassi espressioni	**Valori**	**Tipi**
`Exp ::= ...`	`v::= ...`	τ `::= ...`
`| [l1:e1, ... ln:en]`	`| [l1:v1, ... ln:vn]`	`| [l1:`τ_1`, ... ln:`τ_n`]`
`| e.l`		

Semantica statica:

$$\frac{\forall i : \Gamma \vdash e_i : \tau_i}{[l_1 : e_1, \cdots, l_k : e_k] : [l_1 : \tau_1, \cdots, l_k : \tau_k]} \qquad \frac{\Gamma \vdash e : [l_1 : \tau_1, \cdots, l_k : \tau_k], 1 \leq j \leq k}{\Gamma \vdash e.l_j : \tau_j}$$

Semantica dinamica:

$$\frac{1 \leq j \leq k}{[l_1 : v_1, \cdots, l_k : v_k].l_j \to v_j} \qquad \frac{e \to e'}{e.l \to e'.l}$$

$$\frac{e_j \to e'_j}{[l_1 : e_1, \cdots, l_j : e_j, \cdots, l_k : e_k] \to [l_1 : e_1, \cdots, l_j : e'_j, \cdots, l_k : e_k]}$$

1.2.7 Record: simulazione OCaml

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

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

```

Funzioni di valutazione:

```
    let rec lookupRecord body (Lab l) =
    match body with
        | [] -> raise FieldNotFound
        | (Lab l', v)::t ->
            if l = l' then v else lookupRecord t (Lab l)

```

Interprete:

```
  let rec eval e = match e with
    ...
    | Record(body) -> Record(evalRecord body)
    | Select(e, l) -> match eval e with
        |Record(body) -> lookupRecord body l
        | _ -> raise TypeMismatch

    and evalRecord body = match body with
        |[] -> []
        |(Lab l, e)::t -> (Lab l, eval e)::evalRecord t

```

1.2.8 Array

Un **array** è una *collezione di dati omogenei*; si può vedere come una funzione dal tipo dell'indice (usualmente discreto, e.g. int) al tipo degli elementi. L'implementazione standard consiste nella memorizzazione degli elementi in **locazioni contigue**.

Ovviamente la maggior parte di ciò è falso per gli array di js, che sono implementati in modo non-standard, possono essere non omogenei, sono dinamici e restituiscono "undefined" se si va out-of-bound.

Il controllo sui bound è solitamente lasciato all'utente (contra: Java, Python), anche se Tony Hoare dice (e c'ha ragione) che è meglio fare un vero controllo a run-time, se no ci sono un sacco di vulnerabilità (code reuse, injection).