Linguaggi di Programmazione

Corso di Laurea in "Informatica"

Astrarre sui dati

Valeria Carofiglio

(Questo materiale è una rivisitazione del materiale prodotto da Nicola Fanizzi)

Obiettivi

- tipi di dato astratti
- information hiding
- modularizzazione

Incapsulamento e tipi

- macchina fisica:
  - stringa di bit (unico tipo di dato)
- macchina astratta (LdP ad alto livello):
  - Organizzazione delle stringhe di bit
    - Ogni stringa di bit (valore) ha una capsula (il suo tipo) che la riveste (operazioni associate)
      - LINGUAGGI SICURI RISPETTO AI TIPI: **Capsula Opaca** (accesso alla rappresentazione, consentito solo per mezzo della capsula)

Costruttori di tipo

- I comuni meccanismi di costruzione di tipo (es. array, struct) consentono la definizione di operazioni sui valori che si rappresentano ma non garantiscono che siano le sole modalità di manipolazione
  - es pila di interi realizzata con un array (Pascal, C, ...)
    - operazioni
      - push, pop, empty, top
    - nulla vieta la manipolazione diretta dell'array

Costruttori di tipo

- I comuni meccanismi di costruzione di tipo (es. array, di
  opera o ma
  non g ità di
  manip
  - es p cal, C, ...)
    - o
    - nulla vieta la manipolazione diretta dell'array

Il linguaggio fornisce ***tipi predefiniti*** che nascondono l'implementazione (***astrazione sui dati***)

No per il programmatore

Astrazione sui dati

- Per rendere inaccessibile la rappresentazione: meccanismo ADT
- tipo di dato astratto (ADT- aspetti principali):
  - **nome** tipo
  - implementazione (rappresentazione) **valori**
  - insieme di **nomi** di **operazioni**
  - per ogni operazione
    - **implementazione** dell'**operazione** che usi la rappresentazione fornita
  - **capsula** che separi nomi dei tipi ed operazioni dalla loro implementazione
- la capsula (opaca) separa interfaccia (segnatura) dalla implementazione (realizzazione)
- nei linguaggi più evoluti ADT come tipi predefiniti (ML, CLU, ...)

ADT pila di interi

(capsula opaca)

Rappresentazionenome

nome

```
abstype Int_Stack;

   type Int_Stack = struct {
        int P[100];
        int n; int top;    }

signature
Int_Stack crea_pila();
Int_Stack top(Int_Stack s);
bool empty(Int_Stack s);
Int_Stack push(Int_Stack s, int k);
Int_Stack pop(Int_Stack s);

operations
Int_Stack top(Int_Stack s) {
  return s.P[s.top];
}

bool empty(Int_Stack s) {
  return (s.top == 0);
}
```

Nomi e tipi di operazioni

Implementazione delle op.

```
Int_Stack crea_pila() {
  Int_Stack s = new Int_Stack();
  s.top = 0;
  return s;
}

Int_Stack push(Int_Stack s, int k)
   {
  if (s.top == 100) errore();
  s.P[s.top] = k;
  s.top = s.top + 1;
  return s;
}

Int_Stack pop(Int_Stack s) {
  if (s.top == 0) errore();
  s.P[s.top] = k;
  s.top = s.top - 1;
  return s;
}
```

Fuori dalla rappresentazione del tipo
non c'e alcuna relazione tra tipo astratto e tipo concreto

information hiding

- separare interfaccia da implementazione
  - astrazione sul controllo: nasconde codice del corpo mostra la sua interfaccia (es. funzioni)
  - astrazione dati: nasconde codice ma anche rappresentazione del dato
    - Il sistema dei tipi garantisce la non violazione dell'astrazione
  - vantaggi:
    - possibile sostituire un'implementazione a parità d'interfaccia
      - es. pila realizzata con lista concatenata anziché con vettore

ADT: specifica

- descrizione della semantica delle operazioni di un ADT
  - espresse come relazioni generali astratte (no tipo concreto)
    - in linguaggio naturale, schemi semi-formali, linguaggi formalizzati manipolabili da theorem-prover, ...
    - es. ADT pila di interi

      - **crea_pila**: crea una pila vuota
      - **push**: inserisce un elemento sulla pila;
      - **top**: restituisce l'elemento in cima alla pila (non vuota) senza modificare la stessa
      - **pop**: elimina l'elemento in cima alla pila (non vuota)
      - **empty**: vero sse pila vuota

- specifica = contratto cliente con ADT
  - implementazione corretta deve corrispondere alla specifica (assicurata dalla specifica)

indipendenza dalla rappresentazione

- Due implementazioni corrette di una stessa specifica di un ADT risultano indistinguibili da parte dei clienti
  - Un cliente dell'ADT non deve accorgersi in caso di cambiamento dell'implementazione
- versione debole
  - sostituzione (senza errori di tipo) a parità di segnatura
    - dimostrata come un teorema
      per linguaggi type-safe come CLU, ML, ...

moduli

- ADT → programmazione "in piccolo"
- programmazione "in grande"
  - modulo: astrazione di più strutture dati correlate:
    - es. package Java; unit Pascal/Delphi
  - partizionamento statico di un programma complesso in unità più semplici, dotate di
  - dichiarazioni
    - dati: tipi, variabili, ...
    - operazioni: funzioni, ...
  - regole di visibilità che realizzano:
    - incapsulamento,
    - occultamento dell'informazione (information hiding)

ADT vs. moduli

- in teoria: nessuna differenza
  - ADT è un caso particolare di modulo

- in pratica: moduli più flessibili
  - Grado di permeabilità della capsula
    - Possibilità di selezione degli operatori visibili e del loro grado di visibilità
  - moduli generici (polimorfi)
    - risoluzione al momento dell'uso
  - compilazione separata (unità di compilazione)

esempio
modulo Buffer

Visibile →

Non Visibile →

```
module Buffer imports Counter;
public
        type Buf;
        void insert(reference Buf b, int n);
        int get(Buf b);
        Count c; // quante volte si è usato il buffer
private imports Queue;
        type Buf = Queue;
        void insert(reference Buf b, int n) {
             inqueue(b,n); inc(c);
        }
  int get(Buf b) {
             inc(c); return dequeue(b);
  }
  init_counter(c); // inizializzazione del modulo
}
```

esempio
modulo Counter

```
module Buffer imports Counter;
public
        type Count;
        void init_counter(reference Count c);
        int get(Count c);
        void inc(reference Count c);
private imports Queue;
        type Count = int;
        void init_counter(reference Count c) { c = 0; }
        int get(Count c) {
             return c;
  }
        void inc(reference Count c); {
             c = c+1;
        }
}
```

esempio
modulo Queue

```
module Queue;
public
        type Queue;
        void inqueue(reference Queue q, int n);
        int dequeue(Queue q);
...
private
        ...
        void bookkeep(reference Queue q) {
        ...
  }
...
}
```

modulo

- composizione
  - parte pubblica: visibile all'esterno
  - parte privata: invisibile
- uso di un modulo: importazione
- spesso congiunto con il polimorfismo parametrico
  - risolto a linking-time
    - collegamento a funzioni precompilate di libreria

esempio
modulo Buffer<T>

```
module Buffer<T> imports Counter;
public
        type Buf;
        void insert(reference Buf b, <T> n);
        <T> get(Buf b);
        Count c; // quante volte si è usato il buffer
private imports Queue;
        type Buf = Queue;
        void insert(reference Buf b, <T> n) {
             inqueue(b,n); inc(c);
        }
  <T> get(Buf b) {
             inc(c); return dequeue(b);
  }
  init_counter(c); // inizializzazione del modulo
}
```

esempio
modulo Queue<S>

```
module Queue<S>;
public
        type Queue;
        void inqueue(reference Queue q, <S> n);
        <T> dequeue(Queue q);
...
private
        ...
        void bookkeep(reference Queue q) {
        ...
  }
...
}
```

pila di interi in pseudo C++

```
type Int_Stack = struct {
  int P[100];
  int top;
}

Int_Stack crea_pila() {
  Int_Stack s = new Int_Stack();
  s.top = 0;
  return s;
}

Int_Stack top(Int_Stack s) {
  return s.P[s.top];
}

bool empty(Int_Stack s) {
  return (s.top == 0);
}
```

```
Int_Stack push(Int_Stack s, int k)
  {
  if (s.top == 100) errore();
  s.P[s.top] = k;
  s.top = s.top + 1;
  return s;
}

Int_Stack pop(Int_Stack s) {
  if (s.top == 0) errore();
  s.P[s.top] = k;
  s.top = s.top - 1;
  return s;
}
```