

Sommario: 12 aprile, 2017

- Tipi Astratti: Struttura, API e ADT
- Un ADT per IntStack
- ADT: Modifichiamo l'implementazione
- Un API tante ADT
- ADT con polimorfismo: Definiamo Seq(t)
- Moduli

- **Tipo di Dato Astratto:** Una Collezione di Valori definita dall'insieme delle sole operazioni che possono essere usate per introdurli e operare con essi.
- **Caratteristiche Distintive** sono le operazioni con cui sono completamente definiti e che si dividono in:
 - **Costruttori:** operazioni per introdurre tali valori (come Valori Calcolabili);
 - **Produttori:** operazioni che calcolano tali valori (come Valori Calcolabili);
 - **Modificatori:** operazioni che modificano (componenti di) tali valori (solo per Tipi Astratti modificabili e strutturati);
 - **Osservatori:** operazioni che accedono ai componenti di tali valori (solo per Tipi Astratti strutturati);

Tipi Astratti: Principi/2

- **Tipo di Dato Astratto:** Una Collezione di Valori definita dall'insieme delle sole operazioni che possono essere usate per introdurli e operare con essi.
- **Caratteristiche Distintive** sono le operazioni con cui sono completamente definiti e che si dividono in:...
- **Rappresentazione** di tali valori **non è visibile**, e **nemmeno utile** per programmare con tali valori
- **Implementazione** delle operazioni **non è visibile**¹, e **nemmeno utile** per programmare con tali valori
- **Rappresentazione e Implementazione** possono essere cambiate senza che il comportamento del programma cambi
- **Unità di Programmazione per Dati** della Programmazione Strutturata: Introducono nuove collezioni di valori **nascondendo dettagli implementativi che restano però localizzati** nella definizione del tipo astratto.

¹all'esterno della definizione

Tipi Astratti: API & ADT

- **Tipo di Dato Astratto:** Una Collezione di Valori definita dall'insieme delle sole operazioni che possono essere usate per introdurli e operare con essi.
- Un tipo astratto è una definizione di tipo con la seguente struttura generale:
 - **typeName:** che fornisce il nome del nuovo tipo
 - **signature:** che contiene la segnatura di tutte le operazioni visibili e quindi utilizzabili per il nuovo tipo
 - **implementazione:** che contiene l'intera implementazione (rappresentazione e implementazione op.)
- In una sintassi concreta ispirata a Standard ML:

```
abstype typeName{
  private section: Definizioni di tipo
                  per la rappresentazione
  signature
  public section: Nomi e tipi delle
                 operazioni utilizzabili all'esterno
  operations
  private section: Definizioni delle operazioni,
                  incluse le ausiliarie
}
```

- **Applicative Programming Interface** = typeName + signature

Un ADT IntStack: Stacks bounded di int

```
abstype Int_Stack{
  type Int_Stack = struct{
      int P[100];
      int n;
      int top;
  }
  signature
  Int_Stack create_stack();
  Int_Stack push(Int_Stack s, int k);
  int top(Int_Stack s);
  Int_Stack pop(Int_Stack s);
  bool empty(Int_Stack s);
  operations
  Int_Stack create_stack(){
      Int_Stack s = new Int_Stack();
      s.n = 0;
      s.top = 0;
      return s;
  }
  Int_Stack push(Int_Stack s, int k){
      if (s.n == 100) error;
      s.n = s.n + 1;
      s.P[s.top] = k;
      s.top = s.top + 1;
      return s;
  }
  int top(Int_Stack s){
      return s.P[s.top];
  }
  Int_Stack pop(Int_Stack s){
      if (s.n == 0) error;
      s.n = s.n - 1;
      s.top = s.top - 1;
      return s;
  }
  bool empty(Int_Stack s){
      return (s.n == 0);
  }
}
```

Un rifrasamento in C: Stacks bounded di int

```
1 //abtype IntStack{
//abtype IntStack{
    struct stackStruct{
        int p[100];
        int n;
        int top;
    };
    typedef struct stackStruct stackElem;
    typedef stackElem * IntStack;
/*
    signature
        IntStack createStack();
        IntStack push(IntStack s, int k);
        int top(IntStack s);
        IntStack pop(IntStack s);
        bool empty(IntStack s);

    operations
*/
IntStack createStack(){
    IntStack s = (IntStack) malloc(sizeof(stackElem));
    s->n = 0;
    s->top = 0;
    return s;
}
IntStack push(IntStack s, int k){
    //if (s->n == 100)error
    s->n = s->n+1;
    s->p[s->top] = k;
    s->top = s->top+1;
    return s;
}
//....
```

Accesso della sola signature (interfaccia)

Tipo Astratto: Rappresentazione dei valori e implementazione delle operazioni sono accessibili solo all'interno della definizione.

- Si consideri il seguente frammento di programma di un linguaggio a blocchi, scope statico, ADT e struttura C-like di comandi ed espressioni. Il frammento utilizza lo ADT `Int_Stack` e contiene alcuni errori. Quali e Perché?

```
abstype Int_Stack{...};
int k = 0;
Int_Stack stack1, stack2;
stack1 = push(5, push(1, create_stack()));
stack2 = stack1;
if (stack1.n>0) stack1.p[top - 1] = stack1.p[top];
while (!empty(stack2)){
    k = k + top(stack2);
    pop(stack2);
    stack2.top = k;
}
```

Valori Astratti. Insieme **A** dei valori esprimibili con un ADT.

Esempio

A_{intStack} è insieme di sequenze non-modificabili della forma $[x_1, \dots, x_k]$, quando $k > 0$, $[],$ altrimenti. L'intero x_k è l'ultimo inserito ed è il primo che può essere acceduto.

Stati Concreti. Insieme **C** degli stati definibili nell'ADT per rappresentare A.

Esempio: **abstype** Int_Stack{
 type Int_Stack = struct{
 int P[100];
 int n;
 int top;
 }

A_{intStack} è insieme di tutte le triple $\text{int}[100] \times \text{int} \times \text{int}$

Funzione di Astrazione È una funzione **AF**: **C** \rightarrow **A**

Esempio

$AF(c) = []$ if $c.n == 0$

$AF(c) = [x_1, \dots, x_k]$ con $x_i == c.p[i - 1](\forall i \in [1..n]),$ if $c.n > 0$

Invariate di Rappresentazione È un predicato che individua gli stati legali di **C**

Esempio

$I(c) = (c.n \geq 0) \& (c.n = c.top) \& (c.n \leq 100)$

ADT con Polimorfismo: Signature di Seq<T>

```
abstype Seq<T>
begin
  type
  signature
    function empty():Seq<T>;
    function add(Seq<T>,<T>):Seq<T>;
    function append(Seq<T>,Seq<T>):Seq<T>;
    function size(Seq<T>):integer;
    function at(Seq<T>,integer)<T>;
    function readSeq():Seq<T>
  operations
end;
```

ADT con Polimorfismo: Operazioni di Seq<T>/1

```
abstype Seq<T>
begin
  type
    Seq<T> = ^seqElem;
    seqElem = record body:Seq<T>; tailVal:<T> end;
  signature
  operations
    function empty():Seq<T>;
      begin empty:= null; end;
    function add(u:Seq<T>;x:<T>):Seq<T>;
      var r:Seq<T>;
      begin
        new(r);
        r^.body:= u;
        r^.tailVal:=x;
        add:= r;
      end;
    function append(Seq<T> u,w);
      begin
        if(size(u)=0) then append:=w
        else begin
          if (size(w)=0) then append:=u
          else append:= add(append(u,w^.body),w^.tailVal)
          end;
        end;
      end;
end;
```

... implementazione delle altre operazioni - segue ...

ADT con Polimorfismo: Operazioni di Seq<T>/2

```
abstype Seq<T>
begin
  type
    Seq<T> = ^seqElem;
    seqElem = record body:Seq<T>; tailVal:<T> end;
  signature
  operations
    ...
    function size(Seq<T>):integer;
    begin
      if (u=null) then size:=0
      else size:=1+size(u^.body);
    end;
    function at(Seq<T> u, integer n):<T>;
      /* indefinita se n<0 oppure n>size(u) */
    function innerAt(intSeq u, int k):<T>;
    begin
      if (k=0) then innerAt:= u^taiVal
      else innerAt:= innerAt(u^.body,k-1);
    end;
    begin
      if (n>=0 & n<=size(u))
      then at:= innerAt(u,size(u)-n);
    end;
    function readSeq():Seq<T>;
      /* ausiliaria per I/O */
      /* sequenza interi seguiti da ',' con '[' delimitatore sinistro,
      e un qualunque carattere come delimitatore destro */
    begin ... end;
    procedure writeSeq(Seq<T> u);
      /* ausiliaria per I/O */
    begin ... end;
end;
```

- ADT = Unità di Programmazione per Programmazione in Piccolo
 - operano su un tipo localizzandone la definizione e
 - ne limitano la visibilità in accordo alla regola della "sola segnatura"

- Moduli = Unità di Programmazione per Programmazione in Grande
 - operano su partizioni di sistemi di grandi dimensioni
 - trattandole come parti autonome e
 - compilabili in modo indipendente e
 - con caratteristiche di modificabilità simili agli ADT
 - raggruppano più dichiarazioni (dati e/o funzioni), e
 - definiscono regole di visibilità per tali dichiarazioni
 - Stabilendo ciò che è pubblico e ciò che non lo è
 - importando da moduli ed esportando solo su altri

Modulo: Costrutti Imports, Public, Private

```
module Buffer imports Counter{
  public
    type Buf;
    void insert(reference Buf f, int n);
    int get(Buf b);
    Count c; // how many times buffer has been used
  private imports Queue{
    type Buf = Queue;
    void insert(reference Buf b, int n){
      inqueue(b,n);
      inc(c);
    }
    int get(Buf b){
      return dequeue(b);
      inc(c);
    }
    init_counter(c); // module initialisation part
  }
}
module Counter{
  public
    type Count;
    void init_counter(reference Count c);
    int get(Count c);
    void inc(reference Count c);
  private
    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;
    }
}
```