Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2018/2019

2019

Indice

10. I tipi di dato astratti

10.1. Nozione di tipo astratto

La nozione di algoritmo è indipendente dalla sua codifica nel linguaggio di programmazione; analogamente, con *tipo di dato astratto* (o struttura dati astratta) si intende la specifica di un tipo di dato indipendente dalla sua implementazione e dal linguaggio di programmazione usato per la sua codifica. Intuitivamente, un tipo di dato astratto è una collezione di elementi su cui è possibile eseguire un insieme prefissato di operazioni.

Un *tipo di dato astratto* è costituito da tre componenti:

- il *dominio di interesse*, ovvero l'insieme degli elementi propri del tipo, ed eventuali *altri domini*, ovvero insiemi necessari ad eseguire le operazioni del tipo
- un insieme di *costanti*, usate per denotare valori particolari del dominio d'interesse
- un insieme di *funzioni*, rappresentative delle operazioni proprie del tipo, che operano sugli elementi del dominio di interesse, utilizzando, ove necessario, elementi degli eventuali altri domini.

Un tipo di dato astratto rappresenta la specifica matematica di un insieme di dati e delle operazioni ad esso associate. In quanto tale, essa è indipendente dalla modalità di rappresentazione dei dati e, a maggior ragione, dal particolare linguaggio di programmazione usato.

Una caratteristica fondamentale dei tipi di dato astratto è che a partire da essi si può progettare un algoritmo senza dover far riferimento ad una particolare rappresentazione (o implementazione) del tipo di dato

stesso né dello specifico linguaggio di programmazione usato. Questo comporta un'enorme semplificazione del processo di creazione di un programma, in quanto permette al progettista di focalizzarsi sulle operazioni da eseguire (cosa fare) piuttosto che sul modo in cui esse devono essere realizzate (come fare).

10.1.1. Il tipo astratto Booleano

Un semplice esempio di tipo di dato astratto è il tipo *Booleano*. La sua specifica informale è la seguente:

- dominio di interesse: {*vero*, *falso*}
- costanti: TRUE e FALSE, che denotano rispettivamente *vero* e *falso*
- funzioni: and, or, not

Le costanti possono anche essere definite tramite funzioni (senza argomenti). All'occorrenza, nel seguito sfrutteremo anche questa possibilità per definire le costanti di un tipo.

Una volta definite le operazioni primitive di un tipo, esse possono essere usate per definire funzioni più complesse. Si pensi, ad esempio, all'uso di espressioni booleane all'interno di un algoritmo: partendo dalle operazioni primitive del tipo, vengono definite funzioni arbitrariamente complesse.

10.1.2. Tipi di dato astratti comuni

In generale, è possibile specificare tipi di dato astratti di qualsiasi natura. Alcuni di essi, tuttavia, si distingono per la loro generalità, cui consegue una vasta diffusione e grande importanza. I tipi di dato astratto più comunemente utilizzati sono:

1. Lineari: liste, pile, code, insiemi
2. Non lineari: alberi binari, alberi n-ari, grafi

10.1.3. Utilizzi dei tipi astratti

La nozione di tipo astratto consente di concettualizzare:

- i tipi di dato utilizzati nella progettazione di algoritmi e nella realizzazione dei corrispondenti programmi, quali array, liste, insiemi,

pile, code, alberi, grafi ecc.; si usa spesso il termine *struttura di dati* per riferirsi a questi tipi di dato.

- dati di qualsiasi tipo, che si ritengono importanti in una applicazione.

È importante comprendere che un progettista software non è solo interessato a utilizzare tipi di dato consolidati e ampiamente studiati per realizzare programmi, ma anche a ideare nuovi tipi astratti per rappresentare attraverso essi il dominio di interesse di una applicazione.

Consideriamo, ad esempio, un'applicazione in cui sono rilevanti gli studenti iscritti ad un corso universitario. Possiamo pensare a queste informazioni come ad un tipo astratto che ha come dominio l'insieme degli studenti e come operazioni fondamentali, ad esempio, l'assegnazione di un numero di matricola, l'iscrizione ad un certo anno di corso, la scelta di un piano di studi, e così via.

10.2. Specifica di tipi astratti

Per fornire la specifica di un tipo astratto adottiamo un metodo basato sull'uso schematico del linguaggio naturale. Tale metodo, pur non essendo propriamente formale, ci permette di descrivere senza ambiguità i tipi astratti.

TipoAstratto T

Domini

D1 : descrizione del dominio D1

...

Dm : descrizione del dominio Dm

Costanti

C1 : descrizione della costante C1

...

Ck : descrizione della costante Ck

Funzioni

F1 : descrizione della funzione F1

...

Fn : descrizione della funzione Fn

FineTipoAstratto

Le descrizioni dei domini e delle costanti sono fornite in forma sintetica in linguaggio naturale, mentre la descrizione delle funzioni è più articolata. Tipicamente, D1 viene considerato il dominio di interesse.

10.2.1. Specifica di tipi astratti: Booleano

Forniamo, a titolo esemplificativo, la specifica formale del tipo di dato astratto *Booleano*.

TipoAstratto Booleano

Domini

bool : dominio di interesse { *vero*, *falso* }

Costanti

TRUE : il valore *vero*

FALSE : il valore *falso*

and(bool a, bool b) $\mapsto$ bool
pre: nessuna
post: RESULT è il risultato della congiunzione logica tra a e b

or(bool a, bool b) $\mapsto$ bool
pre: nessuna
post: RESULT è il risultato della disgiunzione logica tra a e b

not(bool a) $\mapsto$ bool
pre: nessuna
post: RESULT è il risultato della negazione logica di a

FineTipoAstratto

10.2.2. I tipi astratti come enti matematici

Si noti che la specifica di un tipo astratto non si riferisce in alcun modo né alla rappresentazione dei valori del dominio d'interesse, né al fatto che tali valori saranno poi eventualmente memorizzati nelle variabili del programma.

Le operazioni associate al tipo vengono descritte semplicemente come funzioni matematiche che calcolano valori a fronte di altri valori, e non come meccanismi che modificano variabili.

In altre parole, in fase di concettualizzazione, le operazioni associate ad un tipo astratto sono specificate mediante funzioni matematiche. Nella successiva fase di realizzazione si procederà alla scelta di come tradurre le varie operazioni del tipo, e si potrà quindi realizzare ogni operazione o come una funzione che calcola nuovi valori (in linea con quanto descritto nella specifica) o come una funzione che modifica le variabili che rappresentano i dati sui quali è invocata.

10.3. Implementazione dei tipi di dato astratti

Il tipo di dato astratto deve essere realizzato usando i costrutti del linguaggio di programmazione:

1. rappresentazione dei *domini* usando i *tipi concreti* del linguaggio di programmazione
2. codifica delle *costanti* attraverso i *costrutti* del linguaggio di programmazione
3. realizzazione delle *operazioni* attraverso opportune *funzioni* del linguaggio di programmazione

10.3.1. Scelta dello schema realizzativo

Per *schema realizzativo* si intende la modalit à con cui un tipo astratto è effettivamente implementato. Essenzialmente, scegliere uno *schema realizzativo* corrisponde a prendere le seguenti due decisioni:

- se le funzioni effettuino o meno side-effect sugli elementi del dominio di interesse
- se le strutture dati coinvolte condividano o meno la memoria utilizzata per la rappresentazione degli elementi del tipo

La scelta dello schema realizzativo costituisce una delle scelte più critiche nella realizzazione di un tipo di dato; essa è dettata tanto da considerazioni relative alle prestazioni quanto da considerazioni di natura modellistica, ovvero dipendenti dall'entità che si vuole modellare con il tipo di dato in esame. In particolare, se questo viene usato per modellare *valori* (si pensi ad esempio a numeri complessi, punti del piano

cartesiano, etc.) è tipicamente indicata une realizzazione che non effettui side-effect. Quando invece il tipo è introdotto allo scopo di modellare *entità* (ad es., persone, oggetti fisici, etc.) è normalmente privilegiata una realizzazione con side-effect. Nel primo caso, infatti, appare naturale pensare un valore come immutabile (il valore 3, in quanto valore, non può essere trasformato in 4), mentre nel secondo ci si può ragionevolmente aspettare che un'entità possa cambiare alcune delle sue proprietà (ad es., potremmo cambiare il colore di una macchina). Queste osservazioni costituiscono delle linee guida generali, tuttavia la scelta dello schema realizzativo è particolarmente importante e deve essere valutata accuratamente caso per caso. In questo corso non vengono affrontate nel dettaglio le motivazioni a supporto di ciascuno schema realizzativo ma soltanto le conseguenze, sul piano dell'implementazione, della scelta effettuata.

10.3.2. Realizzazione delle operazioni

La prima scelta da effettuare riguarda il modo in cui si implementano le operazioni del tipo astratto. In base a questa scelta, classifichiamo gli schemi realizzativi come:

- **con side-effect**, nel caso in cui le funzioni effettuino modifiche sugli elementi del dominio di interesse (opportunamente rappresentati) forniti in input

- **funzionali**, nel caso in cui le funzioni restituiscano nuovi elementi come risultato, senza modificare quelli forniti in input.

10.3.3. Realizzazioni con side-effect

Nelle realizzazioni con side-effect le funzioni che realizzano le operazioni del tipo astratto eseguono side-effect sui dati di input.

Generalmente, questo schema realizzativo è più efficiente, in quanto permette di operare su dati già presenti in memoria, senza dover usare risorse (tempo, memoria) per la loro creazione.

Tuttavia, proprio la possibilità di effettuare side-effect richiede particolare attenzione nell'evitare il problema dell'*interferenza*, termine con cui si indica la modifica *indesiderata* di una struttura dati come conseguenza di un'operazione su un'altra struttura dati.

10.3.4. Realizzazioni funzionali

Nello schema realizzativo funzionale, le funzioni che implementano le operazioni del tipo astratto restituiscono *nuovi* valori del tipo come risultato dell'operazione, senza modificare i dati di input.

Le funzioni sono realizzate seguendo la specifica matematica dei tipi astratti di dato, risultando quindi molto eleganti dal punto di vista formale e semplici da usare.

Le realizzazioni funzionali possono però comportare problemi di inefficienza. Infatti, la necessità di restituire sempre nuovi elementi combinata con il vincolo di non eseguire side-effect può richiedere operazioni di copia (eventualmente parziale) delle strutture manipolate.

10.3.5. Condivisione di memoria tra dati

La seconda decisione il progettista deve prendere riguarda la modalità di gestione della memoria utilizzata per la rappresentazione dei dati. Tale decisione è particolarmente significativa quando il metodo di rappresentazione del tipo astratto prevede l'uso di strutture collegate.

Le alternative possibili sono le seguenti:

- **realizzazione senza condivisione di memoria**: le funzioni operano in modo da assicurare che le rappresentazioni di dati diversi non condividano mai memoria;

- **realizzazione con condivisione di memoria**: le funzioni non impediscono che le rappresentazioni di dati diversi condividano memoria.

Nelle realizzazioni con condivisione, le funzioni possono accedere alle strutture che modellano dati diversi da quelli di input. Esiste cioè la possibilità che dati distinti siano rappresentati mediante strutture (ad esempio collegate) che occupano la stessa locazione fisica. Ad esempio, due strutture collegate potrebbero condividere il contenuto a partire da un dato elemento in poi. Le funzioni che accedono ai dati condivisi devono essere realizzate prestando particolare attenzione a non compromettere la consistenza delle strutture che condividono tali dati (problema dell'interferenza).

Generalmente, la condivisione comporta notevoli benefici tanto dal punto di vista dell'efficienza quanto dell'occupazione della memoria.

10.3.6. Schemi realizzativi privilegiati

Per *schema realizzativo* s intende una combinazione di scelte realizzative tra

- funzionali o con side-effect
- con o senza condivisione di memoria

Conseguentemente abbiamo 4 schemi realizzativi possibili:

- con side-effect, senza condivisione di memoria;
- funzionale, senza condivisione di memoria;
- con side-effect, con condivisione di memoria;
- funzionale, con condivisione di memoria.

Solo due schemi sono interessanti dal punto di vista pratico:

- lo schema con side-effect senza condivisione di memoria, per realizzare dati **mutabili**, cioè manipolati da funzioni che effettuano side-effect;
- lo schema funzionale con condivisione di memoria, per realizzare oggetti **immutabili**, cioè manipolati da funzioni che non effettuano mai side-effect.

Gli altri schemi comportano varie difficoltà implementative, non bilanciate da evidenti benefici. Ad esempio, uno schema con side-effect e condivisione di memoria richiede una gestione complessa della memoria al fine di evitare interferenza, senza che questo comporti un significativo risparmio della memoria usata, mentre uno schema funzionale senza condivisione di memoria tipicamente comporta spreco di memoria dovuto all'intutile replicazione di strutture dati uguali.

10.3.7. Implementazione in C del tipo Booleano

Mostriamo ora un esempio di realizzazione del tipo di dato *Booleano* nel linguaggio C. Scegliamo uno schema realizzativo funzionale senza condivisione di memoria, in quanto il tipo rappresenta un'astrazione di valori matematici. Si noti che la realizzazione proposta nel seguito ha solo scopi didattici e non risulta particolarmente conveniente. Rimane consigliato l'uso degli operatori booleani primitivi.

Per il dominio d'interesse, scegliamo di rappresentare:

- il valore *FALSO* con 0
- il valore *VERO* con 1.

Pertanto, rappresentiamo il dominio bool con il tipo int in C (di cui verranno, effettivamente, usati solo 2 valori):

```
typedef int bool;
```

Per le *Costanti* introduciamo la seguente definizione:

```
#define FALSE 0
#define TRUE 1
```

Per le *Operazioni* abbiamo le seguenti definizioni:

```
bool not(bool x){
  if (x == TRUE){
    return FALSE;
  }
  return TRUE;
}

bool and(bool x, bool y){
  if (x == TRUE && y == TRUE){
    return TRUE;
  }
  return FALSE;
}

bool or(bool x, bool y){
  if (x == TRUE || y == TRUE){
    return TRUE;
  }
  return FALSE;
}
```

10.3.8. Osservazioni

- Per uno stesso tipo astratto si possono avere *più implementazioni* diverse.
- Una implementazione potrebbe avere delle *limitazioni* rispetto al tipo di dato astratto.

- Un tipo di dato può essere *parametrico*, cioè definito a partire da uno o più tipi di dato (i tipi che seguono ne sono un esempio).

Nel seguito ometteremo il termine "astratto", a cui abitualmente viene associata una caratterizzazione matematica e ci riferiremo semplicemente ai tipi di dato fornendo per essi una specifica semi-formale e diverse possibili implementazioni.

10.4. Il tipo astratto Insieme

Un *insieme* è una collezione non ordinata di elementi omogenei, senza ripetizioni. Per denotare un insieme si usa abitualmente la notazione parentetica:

- {} denota l'insieme vuoto
- {e1, e2, ...} denota un insieme contenente gli elementi e1, e2, ecc.

Si noti che le notazioni { e1, e2 } e { e2, e1 } denotano lo stesso insieme.

TipoAstratto Insieme(T)

Domini

Ins : dominio di interesse

T : dominio degli elementi dell'insieme

Funzioni

insiemeVuoto() $\mapsto$ Ins
pre: nessuna
post: RESULT è l'insieme vuoto

estVuoto(Ins i) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se i è il valore corrispondente all'insieme vuoto, false altrimenti

inserisci(Ins i, T e) $\mapsto$ Ins
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i aggiungendo l'elemento e; se e appartiene già a i allora j coincide con i

elimina(Ins i, T e) $\mapsto$ Ins
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i eliminando l'elemento e; se e non appartiene a i allora RESULT coincide con i

membro(Ins i, T e) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se l'elemento e appartiene all'insieme i, false altrimenti

FineTipoAstratto

10.4.1. Realizzazione del tipo astratto *Insieme*

Come già discusso, occorre scegliere tra gli schemi realizzativi:

- funzionale con condivisione di memoria
- side-effect senza condivisione di memoria

A seconda dei casi, può essere preferibile l'uno o l'altro. Vediamo due esempi di realizzazione del tipo *Insieme* secondo uno schema con side-effect senza condivisione di memoria. Le due realizzazioni si distinguono per il modo in cui i dati sono rappresentati: nel primo si usa una rappresentazione mediante SCL mentre nel secondo l'insieme è rappresentato mediante array. Rappresentiamo il dominio di interesse tramite il tipo Insieme, la cui definizione dipende dal tipo di rappresentazione scelta.

Rappresentazione mediante SCL Abbiamo la seguente definizione di Insieme:

```
typedef int T; // Cambia a seconda del tipo trattato

struct NodoSCL {
    T info;
    struct NodoSCL* next;
};

typedef struct NodoSCL TipoNodo;
typedef TipoNodo* Insieme;
typedef TipoNodo* IteratoreInsieme;
```

Si osservi che in questo esempio si assume che il tipo degli elementi T sia il tipo intero (prima riga). La realizzazione risultante può essere facilmente adattata ad un qualsiasi altro tipo cambiando esclusivamente la definizione di T mediante typedef.

Con le scelte effettuate sopra, le funzioni che implementano il tipo (secondo lo schema realizzativo scelto) sono le seguenti:

```
Insieme* insiemeVuoto() {
  Insieme* r = (Insieme*) malloc(sizeof(Insieme*));
  *r = NULL;
  return r;
}
```

```
bool estVuoto(Insieme* ins) {
    return *ins == NULL;
}
```

```
void inserisci(Insieme *ins, T e) {
    if (!membro(ins,e)) {
        TipoNodo* n = (TipoNodo*) malloc(sizeof(
    TipoNodo));
        n->info = e;
        n->next = *ins;
        *ins = n;
    }
}
```

```
void elimina(Insieme *ins, T e) {
    if (*ins == NULL){
      return;
    }
    NodoSCL* p = *ins;
    if (p->info == e){
      *ins = p->next;
      free(p);
      return;
    }
    elimina(&((*ins) -> next),e);
}
```

```
bool  membro(Insieme* ins, T e) {
    NodoSCL* p = *ins;
    while (p!=NULL) {
        if (p -> info == e){
          return true;
        }
        p = p->next;
    }
    return false;
}
```

Si osservi che, mentre le funzioni fin qui presentate permettono di manipolare insiemi generici, nessuna di esse permette la visita dell'insieme. In altre parole, il tipo non mette a disposizioni funzioni primitive che permettano di visitare tutti gli elementi contenuti nell'insieme. Ciò è chiaramente possibile conoscendo la rappresentazione del tipo. Ad

esempio, sapendo che l'insieme è rappresentato come SCL e conoscendo la struttura dei suoi nodi, la visita dell'insieme si riduce essenzialmente alla visita di una SCL. Questa modalità di accesso ha tuttavia l'inconveniente di dipendere dalla rappresentazione del tipo e quindi di imporre la riscrittura delle funzioni di visita da parte delle funzioni (o programmi) cliente. Sarebbe, invece, preferibile una soluzione che garantisca piena compatibiltà del codice cliente con qualsiasi implementazione del tipo. Più avanti verrà presentato un approccio di questo tipo.

Rappresentazione mediante array Mostriamo ora una semplice implementazione del tipo *Insieme* rappresentato mediante array di dimensione fissa; assumiamo cioè che gli insiemi trattati non contengano più di numero prefissato di elementi (ad es. 100). La generalizzazione ad insiemi di qualsiasi cardinalità non pone particolari difficoltà.

Definiamo il tipo Insieme come segue:

```
typedef struct {
    int size; // dimensione dell'array
    int nelem; // num. elementi validi
    T*data; // array
} Insieme;
```

Una possibile realizzazione delle funzioni è la seguente:

```
Insieme* insiemeVuoto() {
    Insieme* ins = (Insieme *)malloc(sizeof(Insieme));
    ins->size = 100;
    ins->nelem = 0;
    ins->data = (T*)malloc(ins->size*sizeof(T));
    return ins;
}
```

```
bool estVuoto(Insieme *ins) {
    return ins->nelem == 0;
}
```

```
void inserisci(Insieme *ins, T e) {
    if (!membro(ins,e)) {
        if (ins->nelem < ins->size) {
            ins->data[ins->nelem] = e;
            ins->nelem ++;
        }
        else {
            printf("ERRORE: array pieno\n");
        }
    }
}
```

```
void elimina(Insieme *ins, T e) {
    int i=0;
    while (i<ins->nelem && ins->data[i]!=e)
        i++;
    if (i==ins->nelem) {
        // nessuna operazione, elemento non presente
    }
    else {
        // spostare altri elementi
        for (int j=i; j<ins->nelem-1; j++)
            ins->data[j] = ins->data[j+1];
        ins->nelem --;
    }
}
```

```
bool membro(Insieme *ins, T e) {
    bool r = false;
    for (int i=0; i<ins->nelem && !r; i++) {
        r = ins->data[i]==e;
    }
    return r;
}
```

Considerazioni analoghe al caso di rappresentazione dell'insieme con SCL possono essere fatte per quanto riguarda l'accesso agli elementi dell'insieme. Proponiamo qui di seguito una possibile soluzione al problema.

10.5. Il tipo astratto *Iteratore*

Con il termine *Iteratore* si indica un tipo astratto i cui elementi permettono di accedere in modo sequenziale a tutti gli elementi di una

collezione. Non sempre l'uso di iteratori è necessario quando vengono manipolate collezioni. Questo è possibile, ad esempio, combinando opportunamente le funzioni primitive dei tipi Lista, Pila e Coda.

Il tipo Iteratore viene definiti in modo indipendente dalla collezione e ovviamente dalla rispettiva implementazione; esso è quindi compatibile con diversi tipi di dato lineari (ad esempio, liste, insiemi, pile, code), e in alcuni casi sono anche con dati non-lineari (ad esempio, alberi).

L'ordine con cui vengono visitati gli elementi della collezione dipende dal tipo di collezione. Ad esempio, in una lista l'ordine sarà dal primo elemento all'ultimo, in un insieme l'ordine sarà arbitrario.

Il tipo astratto *Iteratore* è definito come segue:

TipoAstratto Iteratore(C, T)

Domini

It : dominio di interesse

C : dominio delle collezioni su cui applichiamo l'iteratore

T : dominio degli elementi della collezione

Funzioni

crea(C c) $\mapsto$ It
pre: nessuna
post: RESULT è un iteratore per la collezione c inizializzato per puntare ad un primo elemento della collezione

hasNext(It i) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se l'iteratore i punta ad un elemento valido della collezione, false altrimenti

next(It i) $\mapsto$ T
pre: i punta ad un elemento valido
post: RESULT è il valore dell'elemento puntato dall'iteratore i, l'iteratore viene incrementato per puntare ad un prossimo elemento della collezione non ancora visitato

FineTipoAstratto

10.5.1. Realizzazione dell'Iteratore

Sebbene da un punto di vista tecnico un iteratore possa essere realizzato sia con uno schema funzionale che con side-effect, il secondo risulta più appropriato in quanto gli iteratori astraggono entità mutabili, il cui stato cambia dopo la visita di un elemento.

La rappresentazione di un iteratore deve ovviamente includere le informazioni necessarie ad eseguire la visita. Osserviamo che, a questo

scopo, l'informazione minima necessaria risulta essere un riferimento al prossimo elemento da visitare. A seconda della struttura da visitare, possono essere necessarie informazioni aggiuntive.

Vediamo nel seguito l'implementazione di un iteratore secondo schema realizzativo con side-effect che permetta di accedere agli elementi di una collezione rappresentata mediante SCL o array. Realizzeremo l'iteratore per gli insiemi nelle due varianti implementative viste sopra.

Iteratore per il tipo Insieme rappresentato mediante SCL Facendo riferimento all' implementazione del tipo *Insieme* con rappresentazione mediante SCL, implementiamo l'iteratore nel tipo IteratoreInsieme, definito come segue:

```
typedef struct{
  NodoSCL* ptr;
} IteratoreInsieme;
```

Intuitivamente, l'iteratore è un riferimento al primo elemento del sottoinsieme non ancora visitato. Con questa scelta, una possibile implementazione delle funzioni del tipo è la seguente:

```
IteratoreInsieme* creaIteratoreInsieme(Insieme* ins) {
  IteratoreInsieme* r = (IteratoreInsieme*) malloc(
    sizeof(IteratoreInsieme));
  r->ptr = *ins;
  return r;
}
```

```
bool hasNext(IteratoreInsieme* it) {
    return it->ptr!=NULL;
}
```

```
T next(IteratoreInsieme *it) {
    T r = TERRORVALUE;
    if (it->ptr!=NULL) {
        r = it->ptr->info;
        it->ptr = it->ptr->next;
    }
    else
        printf("ERRORE Iteratore non valido.\n");
    return r;
}
```

Iteratore per il tipo Insieme rappresentato mediante array In questo caso è necessario mantenere informazioni aggiuntive rispetto al riferimento all'elemento da restituire, ad esempio il numero di elementi significativi memorizzati nell'array. Scegliamo di mantenere queste informazioni tramite riferimento alla struttura da visitare:

```
typedef struct {
    Insieme* ins; // riferimento all'insieme da
    visitare
    int ptr; // indice del prossimo elemento
} IteratoreInsieme;
```

Intuitivamente, l'iteratore è un riferimento al primo elemento del sottoinsieme non ancora visitato. Con questa scelta, una possibile implementazione delle funzioni del tipo è la seguente:

```
IteratoreInsieme* creaIteratoreInsieme(Insieme *ins) {
    IteratoreInsieme *it = (IteratoreInsieme *)malloc(
    sizeof(IteratoreInsieme));
    it->ins = ins;
    it->ptr = 0;
    return it;
}
```

```
int hasNext(IteratoreInsieme *it) {
    return it->ptr < it->ins->nelem;
}
```

```
T next(IteratoreInsieme *it) {
    T r = TERRORVALUE;
    if (hasNext(it)) {
        r = it->ins->data[it->ptr];
        it->ptr ++;
    }
    else
        printf("ERRORE Iteratore non valido.\n");
    return r;
}
```

10.5.2. Uso dell'Iteratore

Vediamo ora un semplice esempio di uso dell'iteratore. Realizziamo una funzione che permetta di stampare tutti gli elementi contenuti in un insieme (di interi), indipendentemente da come questo sia rappresentato.

```
void stampa(Insieme* ins) {
    IteratoreInsieme* it = creaIteratoreInsieme(ins);
    while (hasNext(it)) {
        T e = next(it);
        printf("%d ", e);
    }
        printf("\n");
}
```

Ad ogni iterazione del ciclo while, la funzione controlla se è presente un elemento da visitare (hasNext): in caso affermativo, tramite la funzione next, l'elemento viene estratto, quindi stampato, ed il riferimento al prossimo elemento viene posizionato sull'elemento seguente; in caso negativo, il ciclo termina.

È facile verificare che il codice sopra mostrato funziona indipendentemente dalla modalità di rappresentazione scelta per gli elementi del tipo.

10.6. Tipo astratto Lista

Una *lista* è una collezione ordinata di dati omogenei. La lista, ed in generale qualunque collezione, rappresenta un classico esempio di *tipo parametrico*, ovvero un tipo di dato che coinvolge uno o altri tipi di dato, ma il cui comportamento è indipendente da esso.

Se TipoInfo è il tipo degli elementi, il dominio del tipo di dato lista è costituito da tutte le sequenze di elementi di tipo TipoInfo. Per denotare una lista si usa abitualmente la notazione parentetica:

- () denota la lista vuota
- (e1 e2 ...) denota una lista il cui primo elemento è e1, il secondo e2, ecc.

Si noti che (e1, e2) ed (e2, e1) denotano liste distinte, in quanto anche l'ordine degli elementi contenuti caratterizza una lista.

La definizione del tipo astratto Lista è la seguente: [1]

TipoAstratto Lista(T)

Domini

Lista : dominio di interesse del tipo

T : dominio degli elementi che formano le liste

Funzioni

listaVuota() $\mapsto$ Lista
pre: nessuna
post: RESULT è la lista vuota

estVuota(Lista l) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se la lista l è vuota, false altrimenti

cons(T e, Lista l) $\mapsto$ Lista
pre: nessuna
post: RESULT è la lista ottenuta da l inserendo e come primo elemento

car(Lista l) $\mapsto$ T
pre: l non è la lista vuota
post: RESULT è il primo elemento di l

cdr(Lista l) $\mapsto$ Lista
pre: l non è la lista vuota
post: RESULT è la lista ottenuta da l eliminando il primo elemento

FineTipoAstratto

[1] I nomi car e cdr in riferimento alle liste erano usati nelle prime implementazioni del LISP (List Processor), un linguaggio funzionale interamente basato su liste, ideato da John McCarty nel 1958. Essi fuorono suggeriti dalle abbreviazioni per "contents of the address part of register number" (car) e "contents of the decrement part of register number" (cdr), utilizzate in riferimento alla macchina IBM 704, su cui i precursori del Lisp furono implementati.

10.6.1. Realizzazione del tipo Lista

Come già discusso, occorre scegliere tra gli schemi realizzativi:

- funzionale con condivisione di memoria
- side-effect senza condivisione di memoria

A seconda dei casi, può essere preferibile l'uno o l'altro. Vediamo due esempi di realizzazione del tipo *Lista* secondo uno schema funzionale con condivisione. Le due realizzazioni si distinguono per il modo in cui i dati sono rappresentati: nel primo abbiamo una rappresentazione mediante SCL mentre nel secondo le liste sono rappresentate mediante array.

Rappresentiamo il dominio di interesse, ovvero l'insieme delle liste, tramite il tipo TipoLista, la cui definizione dipende dalla rappresentazione adottata. Tale scelta ci permetterà, come vedremo a breve, di garantire che le operazioni definite sul tipo astratto siano indipendenti dalla rappresentazione.

Rappresentazione mediante SCL Se decidiamo di rappresentare le liste mediante SCL definiamo TipoLista come un riferimento al primo elemento della SCL.

```
typedef int T;

struct NodoSCL {
  T info;
  struct NodoSCL *next;
};

typedef struct NodoSCL TipoNodo;

typedef TipoNodo* TipoLista;
```

Con questa scelta, le funzioni che implementano il tipo (secondo lo schema realizzativo scelto) sono le seguenti:

```
TipoLista listaVuota() {return NULL;}
```

```
int estVuota(TipoLista l) {return (l==NULL);}
```

```
TipoLista cons(T e, TipoLista l){
  TipoLista nuovo = (TipoLista) malloc(sizeof(TipoNodo
    ));
  nuovo -> info = e;
  nuovo -> next = l;
  return nuovo;
}
```

```
T car(TipoLista l){
  if (l==NULL){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l->info;
}
```

```
TipoLista cdr(TipoLista l){
  if (l==NULL){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l->next;
}
```

Rappresentazione mediante array Se invece decidiamo di rappresentare le liste mediante array, definiamo TipoLista come un record contenente:

- un campo puntatore data usato come riferimento all'array che modella la lista;
- un campo intero n contenente la dimensione dell'array.

La definizione di TipoLista è pertanto la seguente:

```
typedef struct {
  T* data;
  int n;
} TipoLista;
```

Una possibile realizzazione delle funzioni, coerente con le scelte effettuate, è la seguente:

```
TipoLista listaVuota() {
  TipoLista l;
  l.n=0;
  return l;
}
```

```
int estVuota(TipoLista l) {
  return (l.n==0);
}
```

```
TipoLista cons(T e, TipoLista l){
  TipoLista r;
  r.n=l.n+1;
  r.data = (T*) malloc((r.n)*sizeof(T));
  //Copia l in r
  for (int i = 0; i < l.n; i++){
    r.data[i] = l.data[i];
  }
  // Inserisce e in fondo ad r
  r.data[r.n-1]=e;
  return r;
}
```

```
T car(TipoLista l){
  if (l.n==0){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  return l.data[0];
}
```

```
TipoLista cdr(TipoLista l){
  if (l.n==0){
    printf("ERRORE: lista vuota\n");
    exit(1);
  }
  TipoLista r;
  r.n = l.n-1;
  r.data = &(l.data[1]); // Condivisione di memoria
  return r;
}
```

La differenza tra le due implementazioni risiede nella specifica di TipoLista e nel corpo delle funzioni. Infatti, funzioni che, nelle due implementazioni, realizzano la stessa funzione astratta, presentano *esattamente* stessa segnatura (in quanto questa dipende, in ultima analisi, solo dallo schema realizzativo scelto). Come conseguenza di ciò, qualunque funzione che faccia uso di queste funzioni è perfettamente compatibile con entrambe le implementazioni. In altre parole, le funzioni *cliente* del tipo astratto non hanno bisogno di conoscere i dettagli implementativi del tipo stesso. Si noti come questo effetto (desiderabile, in quanto favorente il riuso di codice) sia conseguenza delle scelte effettuate in fase di realizzazione, in particolare, dell'aver definito il tipo generico TipoLista, definito di volta in volta, a seconda della rappresentazione scelta. Mostriamo di seguito un esempio di uso di tali implementazioni.

10.6.2. Esempi di uso della Lista

Definiamo, a partire dall'implementazione del tipo astratto, alcune funzioni sulle liste:

- calcolo della lunghezza della lista (funzione length): data una lista, ne calcola la lunghezza;

```
int length(TipoLista l){
  if (estVuota(l)) return 0;
  return 1 + length(cdr(l));
}
```

- inserimento di un elemento in coda alla lista (funzione append): data una lista ed un elemento da aggiungere, restituisce una nuova

lista ottenuto dalla prima aggiungendovi il nuovo elemento in coda;

```
TipoLista append(TipoLista l, TipoInfo e){
  if(estVuota(l)){
    return cons(e,l);
  }
  return cons(car(l),append(cdr(l),e));
}
```

- concatenazione di due liste (funzione concat): date due liste l1 ed l2, ne restituisce una nuova, ottenuta concatenando ad l1 gli elementi di l2, mantenendone l'ordine;

```
TipoLista concat(TipoLista l1, TipoLista l2){
  if (estVuota(l2)){
    return (l1);
  }
  return (concat(append(l1,car(l2)),cdr(l2)));
}
```

- restituzione dell'elemento in posizione i di una lista (funzione get): data una lista ed un intero i, restituisce l'elemento della lista in posizione i;

```
TipoInfo get(TipoLista l, int i){
  if (i < 0 || estVuota(l)){
    printf("ERRORE: lista vuota o indice fuori
    dai limiti!\n");
    exit(1);
  }
  if (i==0) return car(l);
  return get(cdr(l),i-1);
}
```

- inserimento di un nuovo elemento in posizione i (funzione ins): data una lista, un intero i, e un elemento da inserire, restituisce una nuova lista ottenuta dalla primo aggiungendovi il nuovo elemento in posizione i.

```
TipoLista ins(TipoLista l, int i, TipoInfo e){
  if (i < 0 || (i>0 && estVuota(l))) {
    printf("ERRORE: indice fuori dai limiti!\n");
    exit(1);
  }
  if (i==0) return cons(e,l);
  return (cons(car(l),ins(cdr(l), i-1, e)));
}
```

Come anticipato, tutte le funzioni cliente sopra definite sono indipendenti dalla rappresentazione scelta. Ovviamente lo stesso non può dirsi per le funzioni del tipo, la cui implementazione è fortemente legata alla sua rappresentazione. Si osservi anche che, come detto, le funzioni qui definite sono compatibili esclusivamente con uno schema realizzativo funzionale.

10.7. Tipo astratto Coda

Una *coda* (o *queue*) è una sequenza di elementi omogenei gestiti con politica *first-in-first-out (FIFO)*, ovvero in cui è possibile inserire ed estrarre elementi, garantendo che l'elemento estratto sia quello presente nella coda da più tempo.

La gestione di dati tramite coda permette di elaborare i dati nell'ordine di arrivo.

La specifica del tipo Coda è la seguente.

TipoAstratto Coda(T)

Domini

Coda : dominio di interesse del tipo

T : dominio degli elementi delle code

Funzioni

codaVuota() $\mapsto$ Coda
pre: nessuna
post: RESULT è la coda vuota

estVuota(Coda c) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se c è il valore corrispondente alla coda vuota, false altrimenti corrispondente alla coda vuota, false altrimenti

inCoda(Coda c, T e) $\mapsto$ Coda
pre: nessuna
post: RESULT è la coda ottenuta dalla coda c inserendo l'elemento e, che ne diventa l'ultimo elemento della coda

outCoda(Coda c) $\mapsto$ Coda
pre: c non è la coda vuota
post: RESULT è la coda ottenuta dalla coda c eliminando l'elemento in testa, cioè che tra quelli presenti era stato inserito per primo

primo(Coda c) $\mapsto$ T
pre: c non è la coda vuota
post: RESULT è l'elemento in testa alla coda c, cioè l'elemento che tra quelli presenti in c era stato inserito per primo

FineTipoAstratto

10.7.1. Realizzazione del tipo Coda

Come nel caso delle liste, entrambi gli schemi relizzativi sono plausibili. Vediamo qui due esempi di realizzazione secondo lo schema con side-effect senza condivisione di memoria. Anche in questi casi considereremo le due rappresentazioni dei dati mediante SCL o array.

Rappresentiamo il dominio di interesse tramite il tipo Coda. A tale proposito, valgono esattamente le stesse considerazioni fatte nel caso del tipo astratto Lista.

Rappresentazione mediante SCL Se decidiamo di rappresentare le code mediante SCL definiamo Coda come un riferimento al primo elemento della SCL.

```
typedef int T;

struct NodoSCL {
  T info;
  struct NodoSCL *next;
};

typedef struct NodoSCL TipoNodo;
typedef TipoNodo* Coda;
```

Con questa scelta, le funzioni che implementano il tipo (secondo lo schema realizzativo scelto) sono le seguenti:

```
Coda* codaVuota() {
  return (Coda*) malloc(sizeof(Coda));
}
```

```
bool estVuota(Coda* c) {return (*c==NULL);}
```

```
void inCoda(Coda* c, T e){
  if (*c == NULL){
    *c = (Coda) malloc(sizeof(TipoNodo));
    (*c) -> info = e;
    (*c) -> next = NULL;
    return;
  }
  inCoda(&((*c) -> next), e);
}
```

```
void outCoda(Coda* c){
  if (c == NULL || *c == NULL){
    printf("ERRORE: input NULL o coda vuota");
    exit(1);
  }
  Coda tmp = *c;
  *c = (*c) -> next;
  free(tmp);
}
```

```
T primo(Coda* c){
  if (*c == NULL){
    printf("ERRORE: coda vuota");
    exit(1);
  }
  return (*c)->info;
}
```

Rappresentazione mediante array La rappresentazione mediante array, nel caso con side-effect, soffre di problemi di efficienza, in quanto ogni inserimento o estrazione implica la riallocazione dell'intero array. Per ovviare a ciò adottiamo la tecnica del *raddoppiamento/dimezzamento*. Nel dettaglio, rappresentiamo una coda con un record contenente:

- un campo data che fa riferimento ad un array di elementi di tipo T
- un campo intero size dove memorizziamo la dimensione dell'array data

- un campo intero nelem per memorizzare il numero di elementi della coda.

La coda è rappresentata dai primi nelem elementi di data, dove l'elemento in posizione 0 rappresenta la testa della coda, ovvero il prossimo elemento che sarà estratto, e l'elemento in posizione nelem-1 l'ultimo elemento.

Gestiamo l'array data come segue:

- inizialmente l'array è vuoto, con size e nelem nulli
- quando viene effettuato un inserimento, se data è pieno, allora data viene ridimensionato con una dimensione pari al doppio della dimensione della coda risultante
- quando viene effettuata un'estrazione, se la dimensione della coda risultante è minore di size/2 la dimensione dell'array viene dimezzata.

In questo modo si riduce la frequenza delle riallocazioni (si noti che ciò implica, in generale, spreco di memoria).

Con queste scelte, abbiamo la seguente definizione del tipo Coda:

```
typedef struct {
  T* data; // elemento data[0]: testa della coda
  int size; // dimesione array
  int nelem; // dimensione coda
} Coda;
```

Conseguentemente, una possibile realizzazione delle funzioni è la seguente:

```
Coda* codaVuota() {
  Coda* r = (Coda*) malloc (sizeof(Coda));
  r->data = NULL;
  r->size = 0;
  r->nelem = 0;
  return r;
}
```

```
bool estVuota(Coda* c) {return (c->nelem==0);}
```

```
void inCoda(Coda* c, T e){
  c->nelem++;
  if (c->nelem > c->size){
    // Raddoppiamento dimensione array
    c -> size = 2 * c-> nelem;
    c -> data = (T*) realloc(c->data,c->size*sizeof(T)
    );
  }
  c->data[c->nelem -1] = e;
}
```

```
void outCoda(Coda* c){
  if (c == NULL || c->nelem == 0){
    printf("ERRORE: input NULL o coda vuota");
    exit(1);
  }
  c->nelem--;
  if (c->nelem < c->size/2){
    // Dimezzamento array
    c->size /= 2;
    c->data = (T*) realloc(c->data,c->size*sizeof(T));
  }
  // Copia tutti gli elementi della coda nella
    componente precedente
  for (int i = 1; i <= c->nelem; i++){
    c->data[i-1]=c->data[i];
  }
}
```

```
T primo(Coda* c){
  if (c->nelem == 0){
    printf("ERRORE: coda vuota");
    exit(1);
  }
  return c->data[0];
}
```

È facile vedere anche in questo caso che eventuali funzioni cliente sono perfettamente compatibili con entrambe le implementazioini, indipendentemente dalla modalità di rappresentazione scelta.

10.8. Tipo astratto Pila

Una *pila* (o *stack*) è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la

regola:

> L'elemento che viene eliminato tra quelli presenti nella pila deve essere quello che è stato inserito per ultimo.

Questa politica di accesso viene detta *LIFO* ("*L*ast *I*n, *F*irst *O*ut").

Con una pila si risolvono facilmente i problemi in cui i dati vanno elaborati in ordine inverso dal loro arrivo (dal più recente al più vecchio). Un tipico esempio d'uso della struttura dati pila è quello in cui essa viene utilizzata per la gestione delle invocazioni di funzione, ovvero la pila dei RDA, dove l'invocazione più recente è la porssima che deve essere processata.

Il tipo di dato astratto Pila è definito come segue.

TipoAstratto Pila(T)

Domini

Pila : dominio di interesse del tipo

T : dominio degli elementi delle pile

Funzioni

pilaVuota() $\mapsto$ Pila
pre: nessuna
post: RESULT è la pila vuota

estVuota(Pila p) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se p è il valore corrispondente alla pila vuota, false altrimenti

push(Pila p, T e) $\mapsto$ Pila
pre: nessuna
post: RESULT è la pila ottenuta dalla pila p inserendo l'elemento e, che ne diventa l'elemento affiorante

pop(Pila p) $\mapsto$ Pila
pre: p non è la pila vuota
post: RESULT è la pila ottenuta dalla pila p eliminando l'elemento affiorante

top(Pila p) $\mapsto$ T
pre: p non è la pila vuota
post: RESULT è l'elemento affiorante della pila p

FineTipoAstratto

10.8.1. Realizzazione della Pila

Le pile possono essere rappresentate mediante strutture collegate lineari. La scelta dello schema realizzativo, quanto la modalità di rappresentazione dei dati devono essere valutate caso per caso.

Omettiamo la realizzazione in C del tipo Pila, lasciandola, in tutte le sue varianti come utile esercizio. A tale proposito, si faccia riferimento agli esempi sopra proposti per i tipi *Lista* e *Coda*