Approfondimenti – Capitolo 1

I LINGUAGGI PER BASI DI DATI AD OGGETTI

Diversi sistemi commerciali per basi di dati supportano i meccanismi di astrazione del modello dei dati ad oggetti. Vengono illustrate le caratteristiche principali dei linguaggi di tali sistemi usando il linguaggio Galileo 97, definito e realizzato presso l'Università di Pisa, per i seguenti motivi:

- permette di focalizzare l'attenzione sui concetti, evitando certe complessità dei linguaggi commerciali;
- presenta un sistema di tipi forte e ricco; i linguaggi commerciali si stanno orientando in questa direzione, ma non hanno in genere raggiunto pienamente l'obiettivo;
- è liberamente disponibile per ambienti Windows e Macintosh.

Il linguaggio Galileo 97 non verrà definito qui in maniera completa; per tale definizione rimandiamo il lettore a [Alb97].

Nella prima parte del capitolo verranno descritti gli aspetti principali del linguaggio per trattare i meccanismi d'astrazione del modello a oggetti. Successivamente si mostrerà come trattare altri meccanismi interessanti non ancora adottati dagli attuali sistemi commerciali. Infine, si mostrerà come è trattato il modello ad oggetti nel linguaggio di due prodotti storicamente significativi, O_2 e UniSQL.

1.1 I linguaggi per basi di dati ad oggetti

Un linguaggio per basi di dati ad oggetti è un linguaggio di programmazione che supporta i meccanismi d'astrazione del modello ad oggetti. Tra i linguaggi di questa categoria vi sono numerose differenze, che riguardano in particolare:

1. *Tipizzazione*: i linguaggi presentano un diverso livello di tipizzazione, che va dalla tipizzazione *forte e statica* fino all'assenza di tipizzazione. Si dice che un linguaggio ha un sistema di tipi *forte e statico* quando ogni errore di tipo, quale ad esempio la spedizione di un messaggio ad un oggetto che

non ha il metodo corrispondente, viene rilevato dal compilatore. La tipizzazione forte e statica è estremamente importante, poiché permette di rilevare in maniera automatica una notevole quantità di errori prima ancora che l'applicazione venga effettivamente sperimentata. Questo tipo di controllo automatico è essenziale soprattutto durante la manutenzione di un sistema informatico, quando si rende necessario modificare alcune parti (tipi o procedure) di un sistema complesso e verificare che queste modifiche non introducano errori nel resto del sistema. Il sistema dei tipi di un linguaggio deve essere tuttavia definito con attenzione per evitare che i controlli di tipo finiscano per diventare un intralcio per il programmatore. Il Galileo 97 è tipizzato in modo forte e statico.

2. *Integrazione e potere espressivo*: un linguaggio per basi di dati completo ha costrutti per trattare dati persistenti e temporanei, e costrutti per specificare il flusso del controllo. In alcuni linguaggi esiste una netta separazione tra alcune di queste classi di costrutti, come nel sistema O_2, dove la definizione di interfacce e metodi si effettua usando due linguaggi diversi, mentre in altri linguaggi, come il Galileo 97, tutti i costrutti sono integrati fra loro.
3. *Relazione tra tipi oggetto ed interfacce*: sono possibili due approcci, il tipo con implementazione unica, adottato nel Galileo 97, e il tipo con implementazioni multiple. Questo punto sarà ripreso in seguito.
4. *Relazione tra tipi oggetti e classi*: anche qui sono possibili due approcci, la classe con inserzione automatica, adottata nel Galileo 97, e la classe con inserzione esplicita. Anche questo punto sarà ripreso in seguito.
5. *Persistenza dei valori*: un linguaggio per basi di dati permette di manipolare sia valori persistenti, ovvero valori che risiedono nella base di dati e che sopravvivono al termine della transazione che li genera, sia valori temporanei, ovvero valori che vengono allocati in memoria centrale e vengono deallocati durante la transazione corrente, o quando essa termina. Una distinzione fondamentale tra i diversi linguaggi riguarda la modalità con cui si stabilisce quali sono i valori persistenti. Citiamo ad esempio le seguenti modalità, che possono anche essere combinate tra loro:

   (a) *persistenza per raggiungibilità*: esiste una "radice di persistenza", definita ad esempio dall'insieme di tutte le classi dichiarate nello schema, e sono persistenti i valori raggiungibili da questa radice di persistenza, indipendentemente dal loro tipo; questo è il modello del Galileo 97;
   (b) *persistenza per tipo*: sono persistenti i valori appartenenti a certi tipi;
   (c) *persistenza esplicita*: sono persistenti i valori che sono stati creati con un'apposita operazione di "creazione di oggetto persistente", oppure che sono stati resi persistenti con un'apposita operazione.

   Qualunque sia il modello di persistenza adottato, è comunque opportuno che i valori persistenti e temporanei siano trattati il più possibile in modo omogeneo, per semplificare la programmazione. In particolare sono desiderabili le seguenti caratteristiche, offerte dal Galileo 97.

   (a) *Trasparenza della persistenza*: è opportuno che il codice che opera su di un valore sia indipendente dalla persistenza dello stesso valore. Ad esempio, la spedizione di un messaggio ad un oggetto temporaneo do-

vrebbe essere indistinguibile dalla spedizione di un messaggio ad un oggetto persistente e, se possono esistere valori di un certo tipo temporanei e valori dello stesso tipo persistenti, una funzione scritta per operare sui valori temporanei dovrebbe poter operare anche su quelli persistenti e viceversa. Molti sistemi offrono questa caratteristica, che era invece assente nei sistemi precedenti, in cui le sole operazioni permesse sui valori persistenti erano in genere il caricamento in una variabile temporanea e l'aggiornamento con il contenuto di una variabile temporanea.

(b) *Persistenza per tutti i tipi*: è opportuno che sia possibile rendere persistenti valori di un qualunque tipo; molti sistemi pongono delle limitazioni rispetto a questo fatto.

(c) *Transitività della persistenza*: è opportuno che tutti gli oggetti raggiungibili da un oggetto persistente siano anch'essi persistenti; anche questa caratteristica non è supportata da tutti i sistemi.

6. *Controllo dei fallimenti*: i vari linguaggi offrono meccanismi piuttosto diversi per specificare da programma le azioni alternative da intraprendere in presenza di fallimenti delle applicazioni.

1.2 La struttura del Galileo 97

Il Galileo 97 è un linguaggio interattivo: l'utente si connette ad una base di dati e immette un'*espressione* (ad esempio, **select** Nome **from** Studenti); il sistema analizza l'espressione, la valuta e ne visualizza il risultato (in questo caso, il nome di tutti gli studenti). L'utente può anche immettere una *definizione* (ad esempio, let giovani := Studenti **where** eta $<$ 20) e il sistema modifica la base di dati aggiungendo tale definizione. I dati persistenti, ovvero i dati contenuti nella base di dati, sono quelli definiti in questo modo e quelli raggiungibili a partire da tali dati. Una definizione può introdurre un tipo (ad esempio un tipo oggetto), una classe, una procedura o un qualunque altro valore; lo schema di una base di dati viene quindi costruito immettendo un insieme di definizioni, e allo stesso modo si definiscono le procedure e le applicazioni.

Le espressioni e le definizioni sono le uniche due categorie sintattiche del Galileo 97. In particolare, anche una procedura è un'espressione, costruita combinando altre espressioni. Ogni espressione Galileo 97 ha un tipo (il *tipo statico* dell'espressione), che viene determinato staticamente dal compilatore. Ogni operazione è applicabile solo ad espressioni con un tipo statico opportuno; nell'uso del Galileo 97 è indispensabile avere chiaro in ogni momento qual è il tipo dell'espressione che si sta scrivendo.

Descriviamo ora il linguaggio partendo dal meccanismo delle definizioni e dal sistema dei tipi, per passare poi a descrivere il modo in cui viene supportato il modello ad oggetti.

1.3 Le definizioni

L'utente Galileo 97 modifica l'insieme di definizioni contenute nella base di dati immettendo *definizioni globali*, come nell'esempio seguente.

```
let   type Peso:= int ;        % definizione globale di un tipo %
let   ilMioPeso:= 63 :Peso     % definizione globale di due valori %
and   ilTuoPeso:= 80 :Peso;
```

Una *definizione elementare di tipo* **type** $\mathcal{A} := \mathcal{T}$ associa l'identificatore $\mathcal{A}$ al tipo $\mathcal{T}$, ed una *definizione elementare di valore* $\mathcal{A} := \mathcal{E}$ associa l'identificatore $\mathcal{A}$ al valore dell'espressione $\mathcal{E}$. Più definizioni elementari collegate con and formano un ambiente $\mathcal{R}$, ed una definizione globale si scrive **let** $\mathcal{R}$;.

Si osservi come la base di dati Galileo 97 può contenere sia definizioni di tipo che definizioni di valori di qualunque tipo.

1.4 Il sistema dei tipi

I tipi primitivi comprendono **int**, **real**, **bool**, **string** e **null**. Le costanti **true** e **false** denotano gli unici valori di tipo **bool**. I valori di tipo **string** sono sequenze di caratteri racchiuse fra doppi apici (" "), **nil** è l'unico valore di tipo **null**.

Gli operatori associati ai tipi primitivi **int**, **real**, **bool** e **string** sono quelli usuali, mentre al tipo **null** è associato solo l'operatore di uguaglianza.

Nuovi tipi possono essere definiti usando opportuni operatori di tipo (o *costruttori* di tipo). Essi sono: coppia, record, valore modificabile, sequenza, unione discriminata, funzione e oggetto. Prima di vedere come si trattano gli oggetti, vediamo brevemente gli altri tipi strutturati (i *tipi strutturati concreti*): tipi record, valori modificabili, sequenze e funzioni.

1.4.1 Record

Un tipo record è un insieme di coppie (etichetta, tipo di valore associato), eventualmente vuoto, dove le etichette ($A_1,\ldots,A_n$) sono tutte diverse:

$[\ A_1 : \mathcal{T}_1; \ldots; A_n : \mathcal{T}_n\ ]$

Un valore record è denotato con:

$[\ A_1 := \mathcal{E}_1; \ldots; A_n := \mathcal{E}_n\ ]$

L'ordine delle etichette in un record e in un tipo record è irrilevante.

Due record sono uguali se hanno lo stesso insieme di coppie (etichetta, valore associato), come nell'esempio seguente.

[A := 2; B := "cc"] = [B := "cc"; A := 2]

Un componente di un record si seleziona con l'operatore punto:

$\mathcal{H}$.A

denota il valore associato all'etichetta A nel record denotato da $\mathcal{H}$.

1.4.2 Sequenze

Le sequenze (o liste) sono utilizzate per rappresentare collezioni finite di valori dello stesso tipo, ed hanno, come vedremo, un ruolo importante nel meccanismo delle classi; sono state preferite agli insiemi perché sono più espressive e più semplici dal punto di vista computazionale.

Il tipo di una sequenza i cui elementi hanno tipo $\mathcal{T}$ si indica con seq $\mathcal{T}$.

Due sequenze sono uguali se e solo se hanno la stessa cardinalità e gli elementi sono uguali e nello stesso ordine.

Una sequenza viene costruita racchiudendo tra parentesi graffe una lista di espressioni separate da ";". Per una sequenza vuota è d'obbligo specificare il tipo degli elementi, così da poter determinare il tipo dell'espressione. Ad esempio le espressioni:

```
{ } :seq int ;
{1; 2; 3};
```

denotano sequenze di interi.

Sulle sequenze sono disponibili diversi predicati e operatori fra i quali quelli tipici su liste (first, rest, append, :: (*cons*) ecc.) e su insiemi (union, difference, intersection, emptyseq, isin ecc.). In particolare, emptyseq($\mathcal{Q}$) è un'espressione booleana che ritorna **true** se $\mathcal{Q}$ è una sequenza vuota, e $\mathcal{E}$ **isin** $\mathcal{Q}$ ritorna **true** se $\mathcal{E}$ è un valore che appare nella sequenza $\mathcal{Q}$.

Gli operatori più importanti sono però quelli definiti solo su sequenze di record, perché sono quelli che vengono utilizzati per scrivere le interrogazioni. Li presentiamo dividendoli in *operatori algebrici*, *quantificatori*, get.

Operatori algebrici Gli operatori algebrici **select from**, In, **where** e **times*** si applicano ad una sequenza di record e ne restituiscono un'altra, il che permette di combinarli tra loro a piacere.[1]

L'espressione **select** $\mathcal{E}$ **from** $\mathcal{W}$ ritorna la sequenza ottenuta valutando l'espressione $\mathcal{E}$ una volta per ogni elemento della sequenza di record $\mathcal{W}$. Se $\mathcal{T}$ è il tipo dell'espressione $\mathcal{E}$, allora **select** $\mathcal{E}$ **from** $\mathcal{W}$ ha tipo **seq** $\mathcal{T}$. Gli identificatori dei record in $\mathcal{W}$ possono essere usati in $\mathcal{E}$ per denotare i valori dei campi corrispondenti. Ad esempio, immaginiamo di avere immesso le seguenti definizioni:

```
let Persone :=
    {[ Cognome     := "Rossi" ;
       AnnoNascita := 1942;
       Tel         := {501215; 526576}];
     [ Cognome     := "Bianchi";
       AnnoNascita := 1977;
       Tel         := {}: seq int ];
     [ Cognome     := "Verdi";
       AnnoNascita := 1962;
       Tel         := {527687; 896332}]
```

1. Più precisamente, **In** è l'unico ad accettare in ingresso sequenze qualunque, e non solo di record, mentre **select from** è l'unico che può restituire sequenze qualunque, e non solo di record.

```
    };

let Auto :=
    {[ Targa := "AT315VZ";  Proprietario := "Rossi" ];
     [ Targa := "AB108NT"; Proprietario := "Rossi" ];
     [ Targa := "BP115UV"; Proprietario := "Verdi" ]
    };
```

Per conoscere l'età nel 1980 degli elementi della sequenza Persone si pone:[2]

```
select   (1980 − AnnoNascita)
from     Persone;

> {38; 3; 18} :seq int
```

L'espressione $\mathcal{W}$ **where** $\mathcal{B}$ restituisce la sequenza formata dagli elementi della sequenza di record $\mathcal{W}$ che soddisfano il predicato $\mathcal{B}$. In $\mathcal{B}$ le etichette degli elementi di $\mathcal{W}$ possono essere usate, come identificatori, per denotare i valori dei campi del record analizzato. L'operatore **where** serve quindi per trovare tutti gli elementi di una sequenza che soddisfano una certa condizione, come nell'esempio seguente:

```
Persone where (AnnoNascita > 1959 And Not emptyseq(Tel));

> {[Cognome := "Verdi"; AnnoNascita := 1962; Tel := {527687; 896332}]}
> :seq [ Cognome:string ; AnnoNascita :int ; Tel :seq int ]
```

L'espressione *Ide* In $\mathcal{Q}$ restituisce una sequenza che contiene, per ogni elemento $\mathcal{X}$ della sequenza $\mathcal{Q}$, un record [*Ide* := $\mathcal{X}$], ed ha quindi tipo **seq** [*Ide* :$\mathcal{T}$], se $\mathcal{Q}$ ha tipo **seq** $\mathcal{T}$. Questo operatore è utile quando si vogliono applicare gli altri operatori algebrici, che lavorano solo su sequenze di record, a sequenze di altro tipo (ad esempio, di interi o di stringhe), oppure quando si vuole associare un nome, in una clausola select o **where** , ad ogni record della sequenza, come nel terzo degli esempi sottostanti, dove l'identificatore P denota l'intera persona sotto esame (si noti che In lega più di **where** che lega più di **select from**).

```
x In 2; 3;

> {[ x := 2 ]; [ x := 3 ]} :seq [x :int ]

select   x*x
from     x In {2;4;6;8}
where    x > 3;

> {16; 36; 64} :seq int

select   [Cognome := P.Cognome; Eta := 1980 − P.AnnoNascita]
from     P In Persone
where    P.AnnoNascita < 1970;

> {[Cognome := "Rossi"; Eta := 38]; [Cognome := "Verdi"; Eta := 18]}
> :seq [Cognome:string ; Eta :int ]
```

2. Quando si mostrano le risposte del sistema, le righe che le rappresentano iniziano con >.

L'espressione $\mathcal{W}_1$ **times*** $\mathcal{W}_2$ denota la sequenza di record ottenuti concatenando ogni record di $\mathcal{W}_1$ con tutti quelli di $\mathcal{W}_2$; per concatenazione di due record r_1 e r_2 si intende un record che ha tutte le coppie di r_1 e tutti quelli di r_2, operazione che è ben tipizzata solo se i due record da concatenare non hanno alcuna etichetta in comune. Ad esempio, la seguente espressione prima genera tutte le possibili coppie di persone ed auto (P, A), e poi seleziona quelle in cui il cognome di P è uguale al proprietario di A; questa operazione di prodotto seguito da una selezione delle sole coppie di elementi che sono in qualche senso correlati è nota come *giunzione*.

```
P In Persone times* A In Auto where P.Cognome = A.Proprietario;

> {[ P := [Cognome := "Rossi"; AnnoNascita := 1942; Tel := {501215; 526576}];
>    A := [Targa := "AT315VZ"; Proprietario := "Rossi"]];
>  [ P := [Cognome := "Rossi"; AnnoNascita := 1942; Tel := {501215; 526576}];
>    A := [Targa := "AB108NT"; Proprietario := "Rossi"]];
>  [ P := [Cognome := "Verdi"; AnnoNascita := 1962; Tel := {527687}];
>    A := [Targa := "BP115UV"; Proprietario := "Verdi"]]}
>  :seq [ P :[Cognome:string ; AnnoNascita:int ; Tel:seq int ];
>         A :[Targa:string ; Proprietario:string ]]
```

La sequenza $\mathcal{W}_2$ può dipendere dalle etichette presenti nella sequenza $\mathcal{W}_1$, ottenendo così un *prodotto dipendente*, come nei due esempi seguenti. Nel primo si accoppia ogni persona P con ogni elemento del suo attributo multivalore Tel; si osservi che la sequenza T **In** P.Tel dipende da P.

```
select   [ Cognome := P.Cognome; Telefono := T ]
from     P In Persone times* T In P.Tel;

> {[Cognome := "Rossi"; Telefono := 501215] ;
>  [Cognome := "Rossi"; Telefono := 526576] ;
>  [Cognome := "Verdi";  Telefono := 527687] ;
>  [Cognome := "Verdi";  Telefono := 896332]}
> :seq [Cognome:string ,Telefono :int ]
```

Il prodotto dipendente serve ad accoppiare ogni elemento a di un certo insieme con tutti gli elementi di un insieme $F(a)$ che sono in qualche modo correlati ad a. Questo insieme $F(a)$ può essere ricavato da un attributo multivalore di a, oppure essere estratto da un diverso insieme, come nel seguente esempio, che mostra un modo alternativo di eseguire la giunzione tra persone e auto vista in precedenza. Stavolta, anziché generare l'intero prodotto e poi selezionare le coppie persona-auto, ogni persona viene accoppiata direttamente con le proprie automobili, ottenendo quindi, con un prodotto dipendente, lo stesso risultato che era stato sopra ottenuto con un prodotto seguito da una selezione.

```
P In Persone times* (A In Auto where P.Cognome = A.Proprietario);

> {[ P := [Cognome := "Rossi"; AnnoNascita := 1942; Tel := {501215; 526576}];
>    A := [Targa := "AT315VZ"; Proprietario := "Rossi"]];
>  [ P := [Cognome := "Rossi"; AnnoNascita := 1942; Tel := {501215; 526576}];
>    A := [Targa := "AB108NT"; Proprietario := "Rossi"]];
>  [ P := [Cognome := "Verdi"; AnnoNascita := 1962; Tel := {527687}];
>    A := [Targa := "BP115UV"; Proprietario := "Verdi"]]}
>  :seq [ P :[Cognome:string ; AnnoNascita:int ; Tel:seq int ];
>         A :[Targa:string ; Proprietario:string ]]
```

Nella teoria degli insiemi, dato un insieme A ed una funzione F che associa ad ogni elemento a di A un insieme $F(a)$, il prodotto dipendente di A ed F è definito come l'insieme di tutte le coppie (a, b) tali che $a \in A$ e $b \in F(a)$. Il prodotto cartesiano $A \times B$ è quindi un caso particolare di prodotto dipendente in cui la funzione $F(a)$ vale sempre B per ogni $a \in A$.

Quantificatori I quantificatori **some with**, **each with** corrispondono ai quantificatori logici $\exists$ e $\forall$.

L'espressione **some** $\mathcal{W}$ **with** $\mathcal{B}$ ritorna il valore **true** se e solo se la condizione $\mathcal{B}$ è vera per almeno un elemento della sequenza di record $\mathcal{W}$.

L'espressione **each** $\mathcal{W}$ **with** $\mathcal{B}$ ritorna il valore **true** se e solo se la condizione $\mathcal{B}$ è vera per tutti gli elementi della sequenza di record $\mathcal{W}$.[3]

In entrambi i casi, le etichette degli elementi di $\mathcal{W}$ possono essere usate in $\mathcal{B}$, come identificatori, per denotare il valore del campo corrispondente dell'elemento sotto esame.

La prima interrogazione nell'esempio trova le persone che hanno almeno un telefono con numero 5x xx xx, e la seconda trova quelle che hanno tutti i telefoni con un numero 5x xx xx.

```
Persone where (some T In Tel with 499999 < T And T < 600000);

> {[Cognome := "Rossi"; AnnoNascita := 1942; Tel := {501215; 526576}];
>  [Cognome := "Verdi";  AnnoNascita := 1962; Tel := {527687; 896332}]}
> :seq [Cognome:string ; AnnoNascita:int ; Tel:seq int ]

Persone where (each T In Tel with 499999 < T And T < 600000);

> {[Cognome := "Rossi";    AnnoNascita := 1942; Tel := {501215; 526576}];
>  [Cognome := "Bianchi"; AnnoNascita := 1977; Tel := {}: seq int ]}
> :seq [Cognome:string ; AnnoNascita:int ; Tel:seq int ]
```

Si osservi che l'espressione Persone **where** 499999 < Tel **And** Tel < 600000 non è corretta, poiché l'operatore < opera su interi mentre Tel è una *sequenza* di interi; in genere, per porre delle condizioni su attributi multivalore è necessario usare un quantificatore.

L'operatore get Quando si desidera estrarre un singolo elemento da una sequenza $\mathcal{W}$, l'espressione $\mathcal{W}$ **where** $\mathcal{B}$ non è sufficiente. Infatti, questa espressione ritorna sempre una sequenza; anche quando $\mathcal{B}$ vale per un unico elemento $\mathcal{X}$, $\mathcal{W}$ **where** $\mathcal{B}$ non ritorna $\mathcal{X}$ ma $\{\mathcal{X}\}$.

Ad esempio, la seguente espressione è scorretta perché rossi non viene legato ad un record ma ad un singoletto, ovvero ad una sequenza che contiene esattamente un record.

```
use rossi := Persone where Cognome = "Rossi"
in   rossi.AnnoNascita;

> Type clash in:
> use rossi:= Persone where Cognome = "Rossi"
```

3. Come in logica, se $\mathcal{W}$ è vuota allora **some** $\mathcal{W}$ **with** $\mathcal{B}$ è falso mentre **each** $\mathcal{W}$ **with** $\mathcal{B}$ è vero.

```
> in rossi.AnnoNascita
> Type belongs to an invalid category. Looking for tuple type,
> I found: seq [Cognome:string ; AnnoNascita:int ; Tel:seq int ]
```

Per trasformare un singoletto nel suo unico elemento si usa l'espressione get $\mathcal{Q}$, che restituisce l'unico elemento della sequenza $\mathcal{Q}$ e genera un fallimento se la sequenza non ha esattamente un elemento.

Ad esempio:

```
use rossi := get (Persone where Cognome = "Rossi")
in   rossi.AnnoNascita;

> 1942 : int

use p := get (Persone where AnnoNascita < 1970)
in   p.AnnoNascita;

> Failure : get: too many elements in the sequence
```

1.4.3 Valori modificabili

In un linguaggio con controllo statico e forte dei tipi, che prevede una nozione di sottotipo, una particolare attenzione va posta al trattamento dei valori modificabili. A differenza di altri linguaggi che non adottano particolari accorgimenti per distinguere, ad esempio, campi di record costanti da quelli modificabili, i campi di record visti negli esempi precedenti non sono modificabili, ma costanti. Per dichiarare campi modificabili (e in generale valori modificabili) occorre assegnare ad essi un particolare valore detto *locazione* (indirizzo o riferimento ad una cella di memoria il cui contenuto può essere riassegnato). Una nuova locazione si genera con l'espressione:

var $\mathcal{E}$

La valutazione di var $\mathcal{E}$ ha il seguente effetto: seleziona una cella di memoria inutilizzata, la inizializza con il valore di $\mathcal{E}$, e ritorna come valore la locazione della cella allocata. Se il tipo di $\mathcal{E}$ è $\mathcal{T}$, allora il tipo di var $\mathcal{E}$ è var $\mathcal{T}$.

Due valori di tipo var $\mathcal{T}$ sono uguali se sono la stessa locazione. Ad esempio:

[a := **var** 3] $\neq$ [a := **var** 3]

perché le due valutazioni di **var** 3 restituiscono due locazioni diverse, ma se si associa ad x una specifica locazione eseguendo **let** x := **var** 3;, allora

[a := x] = [a := x];

Per accedere al contenuto di una locazione si usa l'operatore at (prefisso). In un'espressione:

at $\mathcal{L}$

se $\mathcal{L}$ denota una locazione, at $\mathcal{L}$ denota il suo contenuto, cioè il valore che la memoria corrente le associa. Se $\mathcal{L}$ è di tipo var $\mathcal{T}$, il tipo di at $\mathcal{L}$ è $\mathcal{T}$.

Il valore contenuto in una locazione può essere modificato con l'espressione:

$\mathcal{L} <- \mathcal{E}$

il contenuto della locazione denotata da $\mathcal{L}$ è sostituito dalla denotazione di $\mathcal{E}$. La valutazione dell'espressione modifica la memoria e ritorna nil (il valore di tipo null che viene ritornato da quelle espressioni che si valutano solo per il loro effetto sulla memoria). Ad esempio, se ad x si associa il record [a := **var** 3]:

```
let x := [a := var 3];
```

L'attributo del record si modifica con l'espressione:

```
x.a <- (at x.a) + 1;
```

1.4.4 Funzioni

Una funzione viene denotata con la parola chiave **fun** seguita dalla lista dei parametri formali con il relativo tipo, dal tipo del risultato, e da un'espressione che è il corpo della funzione.

fun$(\mathcal{A}_1 : \mathcal{T}_1, \mathcal{A}_2 : \mathcal{T}_2, \ldots, \mathcal{A}_n : \mathcal{T}_n) : \mathcal{T}$ **is** $\mathcal{E}$

L'espressione ha tipo $(\mathcal{T}_1, \ldots, \mathcal{T}_n \rightarrow \mathcal{T})$. Ad esempio, la funzione seguente calcola il quadrato di un numero:

```
fun(x:int ):int is x * x;
```

ed è di tipo **int** $\rightarrow$ **int** .

Se l'elenco dei parametri è vuoto, come in fun() :$\mathcal{T}$ is $\mathcal{E}$, la funzione ha tipo null $\rightarrow \mathcal{T}$.

L'operazione **fun**()... genera una funzione anonima; per darvi un nome si usa una definizione, come per ogni altro valore:

```
let quadrato := fun(x:int ):int is x * x;
```

Per applicare una funzione $\mathcal{F}$ ai parametri attuali $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ si scrive:

$\mathcal{F}(\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n)$

I parametri sono passati per valore e gli identificatori liberi nel corpo di una funzione sono risolti usando la regola dello scoping statico (si rimanda ad un qualunque testo sui linguaggi di programmazione per una spiegazione di questi termini).

Ogni volta che si usa l'operatore fun, si ottiene una funzione "diversa", come illustrato dal seguente esempio:

```
let IdeFun := fun(x:int ):int is x+1;
[ a := IdeFun ] = [ a := IdeFun ]
[ a := fun(x:int ):int is x+1 ] ≠ [ a := fun(x:int ):int is x+1 ]
```

1.5 Tipo oggetto

Un tipo oggetto si definisce in Galileo 97 usando la seguente sintassi:

type $\mathcal{A}$ <–> [$\mathcal{C}_1$:$\mathcal{T}_1$;
...;
$\mathcal{C}_n$:$\mathcal{T}_n$;
$\mathcal{M}_1$:= **meth** (...):$\mathcal{T}_1$ **is** $\mathcal{E}_1$;
...;
$\mathcal{M}_m$:= **meth** (...):$\mathcal{T}_m$ **is** $\mathcal{E}_m$]
[**before** mk($\mathcal{X}$) if $\mathcal{B}$ do $\mathcal{E}$]
[**before** drop($\mathcal{X}$) if $\mathcal{B}$ do $\mathcal{E}$]

Questa definizione specifica sia la struttura dello stato degli oggetti, formata dagli n campi $\mathcal{C}_i$, che i messaggi $\mathcal{M}_j$ accettati da tali oggetti, che il metodo $\mathcal{E}_j$ che implementa ogni messaggio. Il Galileo 97 segue l'approccio del tipo oggetto con implementazione unica.

È possibile rendere un campo $\mathcal{C}_i$ dello stato inaccessibile dall'esterno usando la sintassi **private** $\mathcal{C}_i$:$\mathcal{T}_n$. Il corpo di un metodo può accedere a qualunque campo dell'oggetto, usando la notazione **self**.$\mathcal{C}_i$, e, con la stessa notazione, può accedere anche ai metodi.

Il valore di un campo $\mathcal{C}_i$ di un oggetto $\mathcal{O}$ può essere ottenuto scrivendo $\mathcal{O}.\mathcal{C}_i$. Un metodo di un campo $\mathcal{M}_i$ senza parametri si invoca scrivendo $\mathcal{O}.\mathcal{M}_i$, mentre un metodo con n parametri si invoca con $\mathcal{O}.\mathcal{M}_i(\mathcal{E}_1, \ldots, \mathcal{E}_n)$.

La definizione di un tipo oggetto definisce i seguenti identificatori:

– l'identificatore $\mathcal{A}$ associato al nuovo tipo, diverso da ogni altro tipo e tipo oggetto;
– l'identificatore mk$\mathcal{A}$, che denota il costruttore degli oggetti del tipo $\mathcal{A}$; questa funzione si aspetta come parametro un record che specifica un valore iniziale per ogni campo dello stato, pubblico o nascosto, e genera un nuovo oggetto, inizializzandone i campi come specificato; ogni applicazione del costruttore mk$\mathcal{A}$ restituisce un valore di tipo $\mathcal{A}$ diverso da tutti gli altri costruiti in precedenza (*uguaglianza per identità*);
– l'identificatore drop$\mathcal{A}$, che denota il distruttore degli oggetti di quel tipo. Quando si applica questa funzione ad un oggetto, questo oggetto è marcato come invalido, ed ogni successivo tentativo di accedervi fallisce.

Poiché il Galileo 97 è un linguaggio con gestione automatica dello spazio irraggiungibile (*garbage collection*), il modo migliore per "eliminare" un oggetto è quello di eliminare ogni cammino di accesso che lo raggiunga, piuttosto che utilizzare l'operatore drop$\mathcal{A}$. Vedremo però che questo operatore ha un ruolo importante in congiunzione con il meccanismo delle classi.

La clausola opzionale **before** mk($\mathcal{X}$) if $\mathcal{B}$ do $\mathcal{E}$ definisce un *trigger*, ovvero un'azione $\mathcal{E}$ che il sistema esegue ogni volta che viene eseguita l'operazione mk$\mathcal{A}$ ed è vera la condizione $\mathcal{B}$. L'azione viene valutata prima dell'esecuzione dell'operazione di creazione dell'oggetto e, se essa solleva un fallimento, l'operazione di creazione non viene eseguita (i fallimenti sono descritti nella Sezione 1.11). L'identificatore $\mathcal{X}$ è legato al parametro attuale della funzione mk$\mathcal{A}$ e può essere usato in $\mathcal{B}$ ed in $\mathcal{E}$. Analogamente, la clausola **before** drop definisce un trigger da valutarsi prima di distruggere un oggetto. Questi trigger sono in genere utilizzati per sollevare fallimenti, impedendo la creazione di un oggetto, se i valori forniti alla funzione mk$\mathcal{A}$ non soddisfano determinati vincoli, o la sua distruzione, se essa portasse alla violazione di certi vincoli. È

anche possibile usare il vincolo non per proibire l'esecuzione delle operazioni ma per provocarne altre, ad esempio per fare sì che la rimozione di un oggetto rimuova tutti gli oggetti ad esso associati. Tuttavia, questo particolare uso dei trigger può essere pericoloso, perché il programmatore può facilmente perdere il controllo degli effetti laterali provocati dai trigger, e può anche accadere che diversi trigger entrino in ciclo, provocando ricorsivamente l'uno l'attivazione dell'altro.

L'esempio che segue mostra la definizione del tipo oggetto Persona, con attributi Codice, Nome, AnnoNascita, Telefono e con un metodo di nome Presentati, e la costruzione di un valore di tipo Persona:[4]

```
let rec
type Persona <−>
    [ Codice        :string ;
      Nome          :string ;
      AnnoNascita   :int ;
      Telefono      :[ Abitazione :var string ];
      Presentati    := meth ():string is "Mi chiamo " & self.Nome ]
    before mk(p)
    if      p.AnnoNascita ¡ 1900
    do      failwith "AnnoNascita deve essere >= 1900" ;

let Mario := mkPersona
    ([ Nome         := "Mario Rossi";
       Codice       := "rssmar23h67";
       AnnoNascita  := 1967;
       Telefono     := [ Abitazione := var "06 222444" ] ]);
```

1.6 L'identità degli oggetti

Quando si confrontano due valori di tipo record o sequenza, tali valori sono uguali se e solo se sono uguali in tutte le loro componenti. Quando si confrontano due oggetti, invece, questi oggetti sono uguali se e solo se sono stati costruiti con la stessa esecuzione dell'operatore mk. Questo fatto si spiega in genere dicendo che ogni esecuzione dell'operatore mk crea un oggetto che ha un'*identità* propria diversa dall'identità di qualunque altro oggetto, e che l'uguaglianza tra oggetti opera un confronto per identità.

Inoltre, se un oggetto A fa riferimento, in un campo b, ad un oggetto B, e se l'oggetto B viene modificato in un qualunque modo, un accesso successivo al campo b di A restituisce la nuova versione, modificata, dell'oggetto B. Questo fatto si spiega a volte dicendo che la modifica di un oggetto B ne muta lo stato ma non l'identità, e che il campo b di A fa riferimento non al valore dei campi di B ma all'oggetto B stesso, oppure alla sua identità.

Queste considerazioni non sono sorprendenti da un punto di vista realizzativo, se si pensa all'identità di un oggetto come ad un riferimento a tale oggetto. Queste nozioni vengono però in genere enfatizzate quando si fa riferimento al modello ad oggetti perché esso viene implicitamente confrontato con il mo-

4. & è l'operatore di concatenazione di stringhe.

dello relazionale, dove l'identità non è direttamente supportata, ma deve essere simulata utilizzando le nozioni di *chiave primaria* e di *chiave esterna*.

1.7 Definizione di tipi oggetto per ereditarietà

Un tipo oggetto $\mathcal{T}$ può essere definito per *ereditarietà* da un altro tipo $\mathcal{T}'$, detto *supertipo* di $\mathcal{T}$, come segue:

type $\mathcal{T}$ <–> **is** $\mathcal{T}'$ **and** $\mathcal{H}$
[**before** mk($\mathcal{X}$) if $\mathcal{B}$ **do** $\mathcal{E}$]
[**before** drop($\mathcal{X}$) if $\mathcal{B}$ **do** $\mathcal{E}$]

dove $\mathcal{H}$ specifica quali proprietà (in questo capitolo useremo il termine "proprietà" per riferirci all'insieme dei campi e dei metodi di un oggetto) vanno aggiunte o ridefinite rispetto a $\mathcal{T}'$; $\mathcal{H}$ può essere vuoto ([ ]).

Il tipo $\mathcal{T}$ *eredita* le proprietà del supertipo ed ha in più le proprietà specifiche definite in $\mathcal{H}$.

L'eredità è *stretta*, cioè una proprietà del supertipo $\mathcal{A}_i$, di tipo $\mathcal{T}_i$, può essere ridefinita in $\mathcal{H}$, ma solo specializzandone il tipo, ovvero il nuovo tipo $\mathcal{T}_i'$ di $\mathcal{A}_i$ deve essere un sottotipo di $\mathcal{T}_i$; la nozione di sottotipo è definita in piena generalità nella prossima sezione.

Infine, in una definizione di un tipo oggetto per ereditarietà, si può usare all'interno della nuova definizione di un metodo $\mathcal{M}$ la realizzazione data per esso nel supertipo, usando la sintassi **super**.$\mathcal{M}$(...).

Un esempio di un tipo oggetto definito per ereditarietà è:

```
let rec
type Studente <–> is Persona and
   [ Matricola       :string ;
     CorsoDiLaurea   :string ;
     Telefono        :[ Abitazione :var string ; Dimora :var string ];
     Presentati      := meth ():string is
                     super.Presentati &
                     " Sono iscritto al corso di laurea " &
                     self.CorsoDiLaurea ];
```

Quando un tipo Sotto è definito per ereditarietà da un tipo Sopra, poiché la creazione di un oggetto in Sotto crea lo stesso oggetto anche in Sopra, tale operazione provoca la valutazione anche dei trigger **before** mk del supertipo, per cui tali trigger vengono ereditati. Poiché la rimozione di un oggetto da Sopra ne provoca la rimozione anche da Sotto, in questo caso è il supertipo che, in un certo senso, eredita i trigger **before** drop dal sottotipo. Viceversa, poiché l'operazione dropSotto($\mathcal{O}$) non elimina completamente la possibilità di operare sull'oggetto $\mathcal{O}$, ma, in un certo senso, gli fa solo perdere il tipo Sotto trasformandolo in un valore del supertipo Sopra, questa operazione non provoca la valutazione del trigger **before** drop del supertipo (torneremo in seguito sul significato dell'operazione di drop da un sottotipo).

1.8 Gerarchie di tipi

1.8.1 Le regole di sottotipo

Un tipo $\mathcal{T}$ è sottotipo di un tipo $\mathcal{T}'$ (e scriveremo $\mathcal{T} \subseteq \mathcal{T}'$) quando tutti gli operatori e le funzioni definiti sui valori di tipo $\mathcal{T}'$ possono essere applicati a valori del tipo $\mathcal{T}$. In Galileo 97 la relazione di *sottotipo* è particolarmente ricca e coinvolge non solo i tipi oggetto ma anche tutti i costruttori di tipo concreto.

La relazione di sottotipo per i tipi concreti si basa solo sulla loro struttura, senza bisogno di particolari dichiarazioni da parte del programmatore. La relazione di sottotipo tra tipi oggetto va invece dichiarata esplicitamente, definendo il sottotipo per ereditarietà dal supertipo.[5]

Con riferimento ai tipi visti finora, le regole per stabilire quando due tipi sono nella relazione di sottotipo sono:

1. Se $\mathcal{T}$ e $\mathcal{V}$ sono tipi record, allora $\mathcal{T} \subseteq \mathcal{V} \Leftrightarrow$ l'insieme delle etichette di $\mathcal{T}$ contiene l'insieme delle etichette di $\mathcal{V}$ e, se $\mathcal{T}'$ e $\mathcal{V}'$ sono i tipi di un'etichetta comune, allora $\mathcal{T}' \subseteq \mathcal{V}'$.
   Ad esempio, il tipo

   ```
   [ Nome     :string ;
     Indirizzo :[ Citta:string ] ]
   ```

   è supertipo di

   ```
   [ Nome         :string ;
     Indirizzo    :[ Citta:string ; Strada:string ];
     AnnoNascita  :int ]
   ```

   Questa regola è motivata dal fatto che la sola operazione disponibile sui record è l'estrazione di un campo, per cui, informalmente, un record che metta a disposizione un maggior numero di campi può essere usato in ogni contesto in cui possa essere usato un record con un sottoinsieme di quei campi.
2. **seq** $\mathcal{R} \subseteq$ seq $\mathcal{S} \Leftrightarrow \mathcal{R} \subseteq \mathcal{S}$.
3. **var** $\mathcal{T} \subseteq$ **var** $\mathcal{V} \Leftrightarrow \mathcal{T}$ e $\mathcal{V}$ sono lo stesso tipo.
   Questa regola è particolarmente importante perché stabilisce che fra i tipi di valori modificabili non esiste una relazione generale di sottotipo. Pertanto, nella definizione di tipi oggetto per ereditarietà, una proprietà modificabile del supertipo non può essere ridefinita per specializzazione.
   Quando $\mathcal{T}$ è un sottotipo di $\mathcal{V}$, ma è diverso da $\mathcal{V}$, **var** $\mathcal{T}$ non è un sottotipo di **var** $\mathcal{V}$ a causa dell'operazione di aggiornamento. Infatti, se v è un valore di tipo $\mathcal{V}$, l'operazione x <– v è legittima se x ha tipo **var** $\mathcal{V}$, ma non se x ha tipo **var** $\mathcal{T}$, perché altrimenti si associerebbe la locazione x di tipo **var** $\mathcal{T}$ al valore v che non appartiene al tipo $\mathcal{T}$ ma solo ad un suo supertipo.
   Si osservi l'esempio seguente:

   ```
   let type Indirizzo := [Via:string ; Citta:string ]
   ```

5. Nell'attuale versione del Galileo 97 non sono definibili gerarchie multiple fra i tipi oggetto.

```
ext type Viaggiatore := [Nome:string ; Recapito :var Indirizzo ];
let  Bianchi :=
     [ Nome := "MarioBianchi";
       Recapito := var [Via := "Corso Ferraris"; Citta := "Torino"; Stato := "Italia"]]
and CambioInd := fun(x :Viaggiatore, y :Indirizzo) :null is (x.Recapito) <– y;

CambioInd(Bianchi, [Via := "Via Roma"; Citta := "Parigi"]);

(at Bianchi.Recapito).Stato;
```

In accordo con la regola precedente l'applicazione di CambioInd a Bianchi non è corretta, perché il tipo di Bianchi non è sottotipo di Viaggiatore. Se venisse adottata la regola, scorretta, che **var** $\mathcal{T} \subseteq$ **var** $\mathcal{V}$ se e solo se $\mathcal{T} \subseteq \mathcal{V}$, l'applicazione di CambioInd a Bianchi sarebbe permessa, e la successiva valutazione di (**at** Bianchi.Recapito).Stato causerebbe un errore a tempo di esecuzione, dal momento che l'indirizzo di Bianchi ha perso il campo Stato.

4. $\mathcal{T}_1,\ldots,\mathcal{T}_n \to \mathcal{T} \subseteq \mathcal{T}'_1,\ldots,\mathcal{T}'_n \to \mathcal{T}'$
   $\Leftrightarrow \mathcal{T}'_1 \subseteq \mathcal{T}_1 \wedge \ldots \wedge \mathcal{T}'_n \subseteq \mathcal{T}_n \wedge \mathcal{T} \subseteq \mathcal{T}'$ (regola di *controvarianza* sui tipi degli argomenti di una funzione).

   Si noti l'inversione della relazione sui tipi argomento. Per chiarire questa regola, si consideri il tipo Studente sottotipo di Persona.

   Ogni funzione che ritorna un valore di tipo Studente ritorna anche un valore di tipo Persona: $(\mathcal{T} \to \text{Studente}) \subseteq (\mathcal{T} \to \text{Persona})$.

   Tuttavia una funzione che si aspetta come parametro un valore di tipo Studente, e magari ne estrae la matricola, non può essere applicata ad un valore di tipo Persona; al contrario, ogni funzione con un parametro di tipo Persona può essere applicata ad un valore di tipo Studente, pertanto per ciò che riguarda il tipo del parametro la regola è: $(\text{Persona} \to \mathcal{V}) \subseteq (\text{Studente} \to \mathcal{V})$.

   Per mostrare come una regola diversa sarebbe scorretta, si consideri il seguente esempio:

```
let  type Persona := [Nome:string ]
and type Studente := [Nome:string ; Scuola:string ]
and Franco := [Nome := "Franco"]
and FrancoStudente := [Nome := "Franco"; Scuola := "Universita' di Pisa"]
and ApplicaPersona := fun(g :Persona –> string , x :Persona):string is g(x)
and NomeScuola := fun(x :Studente):string is x.Scuola;

ApplicaPersona(NomeScuola, Franco);
```

   In accordo con la regola precedente l'applicazione di ApplicaPersona non è corretta, perché il tipo di NomeScuola (Studente $\to$ **string**) non è sottotipo di (Persona $\to$ **string**). Se venisse adottata la regola, scorretta, $(\mathcal{T} \to \mathcal{V}) \subseteq (\mathcal{T}' \to \mathcal{V}') \Leftrightarrow \mathcal{T} \subseteq \mathcal{T}' \wedge \mathcal{V} \subseteq \mathcal{V}'$, l'applicazione di ApplicaPersona sarebbe permessa, e si genererebbe un errore a tempo di esecuzione per la selezione del valore della scuola di Franco che non è uno studente.

5. $\mathcal{T} \subseteq \mathcal{T}'$ se $\mathcal{T}$ è un tipo oggetto definito come $\mathcal{T}$ <–> is $\mathcal{T}'$ and $\mathcal{H}$.
6. $\mathcal{T} \subseteq [\Downarrow \mathcal{T}]$ se $\mathcal{T}$ è un tipo oggetto e $\Downarrow \mathcal{T}$, la segnatura di $\mathcal{T}$, è l'insieme di coppie "etichetta :tipo" che descrive l'interfaccia di $\mathcal{T}$. Questa segnatura contiene una coppia $\mathcal{C} : \mathcal{T}$ per ogni campo pubblico o metodo senza parametri $\mathcal{C}$ di tipo $\mathcal{T}$, e una coppia $\mathcal{M} : \mathcal{T}_1,\ldots,\mathcal{T}_n \to \mathcal{T}$ per ogni metodo $\mathcal{M}$ di

tipo $\mathcal{T}$ che ha *n* parametri di tipo $\mathcal{T}_1,\ldots,\mathcal{T}_n$.
Ad esempio, il tipo

```
let rec
type Quadrato <−>
   [ private mioLato  :var int ;
     lato             := meth (): int is at self.mioLato;
     setLato          := meth (l:int ): null is self.mioLato <− l;
     area             := meth (): int is (at self.mioLato) * (at self.mioLato) ];
```

è sottotipo di [⇓Quadrato], ovvero di:

```
[ lato :int ; setLato :int −> null; area :int ]
```

Da questa regola discende il fatto che tutti gli operatori su sequenze di record possono essere applicati anche a sequenze di oggetti.

7. Per ogni tipo $\mathcal{T}$, $\mathcal{T} \subseteq \mathcal{T}$ (riflessività), e $\mathcal{T} \subseteq \mathcal{U}, \mathcal{U} \subseteq \mathcal{V}$ implica $\mathcal{T} \subseteq \mathcal{V}$ (transitività). Dalla transitività si ricava, ad esempio, che se $\mathcal{T}$ è un tipo oggetto, non solo $\mathcal{T} \subseteq [\Downarrow\mathcal{T}]$, ma $\mathcal{T}$ è anche un sottotipo di tutti i tipi record che sono supertipi di $[\Downarrow\mathcal{T}]$.

1.8.2 Ereditarietà e controvarianza

Come specificato nella Sezione 1.7, i metodi di un oggetto possono essere ridefiniti in un sottotipo solo specializzandone il tipo (si osservi che lasciare il tipo immutato è legittimo, poiché ogni tipo è sottotipo di se stesso). Grazie alla regola di controvarianza del sottotipo sui tipi funzione, specializzare il tipo di un metodo significa *specializzare* il tipo del risultato e *generalizzare* il tipo degli argomenti, ovvero sostituire il tipo di alcuni argomenti con un supertipo.

Questa regola è chiamata la *controvarianza del tipo degli argomenti dei metodi*, essa è necessaria per permettere la tipizzazione statica del sistema, ma è considerata poco intuitiva e poco naturale. Si consideri il seguente esempio:

```
let rec
type Persona <−>
        [ Nome   :string ;
          Simile  := meth (altro:Persona) :bool is altro.Nome = self.Nome ];

let rec
type Studente <−> is Persona and
        [ Scuola  :string ;
          Simile  := meth (altro:Studente) :bool is
                        super.Simile(altro) And altro.Scuola = self.Scuola ];
```

La definizione di Studente viene rifiutata dal sistema perché il metodo Simile è ridefinito in modo *covariante*. Se il sistema accettasse questa definizione, sarebbe poi possibile generare un errore a tempo di esecuzione. Si consideri infatti l'esempio sottostante, eseguito in un Galileo 97 immaginario dove la definizione di sopra fosse stata accettata; nell'ultima riga, lo studente ugo riceve, come parametro del metodo Simile, la persona luigi, per cui il metodo Simile di ugo fallisce quando cerca di leggere il campo Scuola di luigi.

```
let luigi      := mkPersona([Nome:= “Luigi”]);
let ugo        := mkStudente([Nome:= “Ugo”; Scuola:= “G. Galilei”]);
```

```
let confronta   := fun(x,y :Persona) :bool is x.Simile(y);
confronta(ugo,luigi);

> Fallimento a tempo di esecuzione:
> campo "Scuola" non trovato
```

D'altra parte i metodi covarianti sono spesso utili, per cui molti sistemi li accettano, pur sapendo che possono provocare la generazione di errori di tipo a tempo di esecuzione.

Sono state di recente proposte delle tecniche che permetterebbero la tipizzazione statica di metodi covarianti, basate sulla tecnica dei "multimetodi", ovvero su messaggi che scelgono il metodo da eseguire dopo avere esaminato il tipo di tutti i parametri ([Ghe91, CGL95]). Usando queste tecniche, la definizione degli studenti sopra fornita sarebbe accettata e non ci sarebbero errori a tempo di esecuzione perché ugo, nello scegliere il metodo da usare, terrebbe conto anche del tipo di luigi, e userebbe quindi il metodo ereditato dal tipo Persona. Queste tecniche non sono però ancora state adottate in nessun sistema commerciale.

1.8.3 Uso della relazione di sottotipo

Se $\mathcal{T} \subseteq \mathcal{T}'$, allora tutte le funzioni e tutti gli operatori definiti sul tipo $\mathcal{T}'$ possono essere applicati a valori di tipo $\mathcal{T}$.

Inoltre, la relazione di sottotipo è utilizzata anche in tutte quelle situazioni in cui si devono combinare più valori dello stesso tipo, come nelle espressioni (a) **if** $\mathcal{B}$ **then** $\mathcal{V}_1$ **else** $\mathcal{V}_2$, (b) $\mathcal{V}_1$ append $\mathcal{V}_2$, (c) $\{\mathcal{V}_1;\ldots;\mathcal{V}_n\}$ (costruzione di una sequenza).

Se ogni $\mathcal{V}_i$ ha tipo $\mathcal{T}_i$, nelle prime due situazioni la regola specifica che l'espressione è corretta se $\mathcal{T}_1 \subseteq \mathcal{T}_2$ oppure se $\mathcal{T}_2 \subseteq \mathcal{T}_1$, e il tipo più generale è anche il tipo dell'intera espressione; una generalizzazione di questa regola viene utilizzata nel terzo caso.

1.9 Interpretazione dei messaggi e semantica degli autoriferimenti

Quando si invia un messaggio m ad un oggetto O occorre risolvere due problemi: (a) quale metodo si usa per rispondere al messaggio e (b) qual è il significato dell'eventuale autoriferimento **self** presente nel corpo del metodo che si usa per rispondere al messaggio.

In un linguaggio ad oggetti la risposta è univoca: ogni oggetto usa i propri metodi per rispondere ai messaggi, e **self** denota l'oggetto che sta rispondendo al messaggio.

Si consideri, ad esempio la seguente funzione:

```
let Stampa := fun(x: Persona):string is x.Presentati;
```

Anche se x è un'espressione di tipo Persona, il messaggio x.Presentati non verrà necessariamente valutato usando il metodo definito nel tipo Persona, ma, per

ogni chiamata della funzione Stampa, si utilizzerà il metodo Presentati del parametro attuale. Questo modo di interpretare i messaggi viene chiamato *late binding* o *binding dinamico* (*dynamic binding*).

Un altro approccio possibile per rispondere ad un messaggio sarebbe quello di usare il metodo relativo al tipo che il compilatore associa all'oggetto destinatario. Questo altro modo di interpretare i messaggi viene chiamato *early binding* o *binding statico* (*static binding*). Secondo questo approccio, la funzione Stampa dell'esempio invocherebbe sempre il metodo Presentati del tipo Persona, indipendentemente dall'oggetto associato ad x.

I linguaggi ad oggetti utilizzano il *late binding* perché questo modo di interpretare i messaggi permette di scrivere codice le cui funzionalità possono essere estese in modo relativamente agevole. Ad esempio, dopo avere definito la funzione Stampa, si supponga di definire un nuovo sottotipo di Persona, con una nuova implementazione per il metodo Presentati. Passando oggetti del nuovo sottotipo alla funzione Stampa, il risultato sarà valutato usando il metodo introdotto per il sottotipo, senza bisogno né di ridefinire né di ricompilare la funzione Stampa. Quindi, grazie al *late binding*, la funzione Stampa si adatta all'evoluzione delle definizioni di sottotipi.

Per quanto riguarda il problema degli autoriferimenti, si considerino le seguenti definizioni di tipi oggetto e le costruzioni di due oggetti v1 e v2:

```
let rec
type Persona <−>
        [ AnnoNascita   :int ;
          Nome          :string ;
          Eta           := meth ():int is AnnoCorrente − self.AnnoNascita;
          Stampa        := meth ():string is
                              "Si chiama " & self.Nome &
                              "ed ha " & stringofint(self.Eta) & " anni." ];
let rec
type PersonaDeceduta <−> is Persona and
        [ AnnoDecesso   :int ;
          Eta           := meth ():int is self.AnnoDecesso − self.AnnoNascita ];

let v1:= mkPersona([ AnnoNascita := 1950; Nome := "Caio" ]);
let v2:= mkPersonaDeceduta([ AnnoNascita := 1950; Nome := "Tizio"; AnnoDecesso := 1970 ]);
```

Supponendo che AnnoCorrente = 1997, i risultati delle seguenti espressioni sono:

```
v1.Eta;      vale 47
v1.Stampa;   vale "Si chiama Caio ed ha 47 anni."
v2.Eta;      vale 20
v2.Stampa;   vale "Si chiama Tizio ed ha 20 anni."
```

Le espressioni v1.Stampa e v2.Stampa illustrano bene il problema degli autoriferimenti: entrambe provocano la valutazione dello stesso metodo, ma producono risultati diversi a causa del messaggio **self**.Eta presente nel metodo. Nel primo caso viene restituito il valore di Eta in v1, mentre nel secondo caso viene restituito il valore di Eta in v2.

I linguaggi ad oggetti prevedono anche un altro particolare tipo di riferimento nel corpo di un metodo, il riferimento a **super**, già introdotto nella Sezione 1.7. Quando il metodo da eseguire per rispondere al messaggio m manda a

sua volta un messaggio n all'oggetto **super**, il destinatario di questo secondo messaggio è sempre l'oggetto che aveva ricevuto il messaggio m, come nel caso di **self**, ma, solo per stabilire quale metodo usare per rispondere ad n, questo oggetto ignora la propria ridefinizione di n, ed utilizza la definizione di n più specifica tra quelle ereditate. L'uso del **super** consente pertanto ad un metodo di usare metodi ereditati anche quando questi ultimi sono stati ridefiniti.

1.10 Classi

Le classi sono un particolare meccanismo per denotare sequenze modificabili di valori, organizzate in gerarchie di sottoinsieme. Le classi quindi si differenziano dalle sequenze per due ragioni:

1. la sequenza denotata da una classe (l'*estensione* della classe) si modifica per effetto di inserzioni e rimozioni di elementi, mentre una sequenza non è modificabile;
2. a partire da una classe si possono definire delle sottoclassi (con un meccanismo definito più avanti) e il sistema mantiene una relazione di sottoinsieme fra l'estensione della classe e quelle delle sottoclassi, per cui, ad esempio, quando si inserisce un elemento in una sottoclasse esso diventa automaticamente un elemento della classe. Fra i tipi sequenza è definita una relazione di sottotipo, ma non una relazione di sottoinsieme fra due valori sequenze.

Le classi servono a modellare le collezioni dell'universo del discorso. Una classe si definisce con il seguente costrutto, dove $\mathcal{T}_r$ si specifica con la sintassi già definita per i tipi oggetto:[6]

$\mathcal{A}$ **class** $\mathcal{T}$ <–> $\mathcal{T}_r$
[**before** mk($\mathcal{X}$) **if** $\mathcal{B}$ **do** $\mathcal{E}$]
[**before** drop($\mathcal{X}$) **if** $\mathcal{B}$ **do** $\mathcal{E}$]
[**key** (ListaCampi)]

Questo costrutto definisce contemporaneamente un tipo oggetto $\mathcal{T}$, con i relativi operatori mk e drop, ed una classe $\mathcal{A}$, che conterrà tutti gli oggetti validi del tipo $\mathcal{T}$. La clausola **key** (ListaCampi), dove ListaCampi è un elenco di campi del tipo $\mathcal{T}$, specifica un vincolo di superchiave: non possono esistere due oggetti diversi nella classe $\mathcal{A}$ che coincidono per ciò che riguarda il valore di ciascuno dei campi in ListaCampi. Il vincolo è verificato al momento della creazione di un oggetto, come il vincolo **before** mk, ed ogni tentativo di costruire un oggetto che lo vìola provoca il fallimento dell'operazione di costruzione.

Il Galileo 97 supporta il modello delle classi con inserzione automatica, per cui l'operatore mk$\mathcal{T}$ inserisce l'oggetto creato nella classe $\mathcal{A}$, e l'operatore drop$\mathcal{T}$, oltre a marcare l'oggetto come non valido, lo elimina dalla classe $\mathcal{A}$.

Nell'esempio che segue è definita la classe Persone, i cui elementi sono oggetti di tipo Persona:

6. In realtà in Galileo 97 si possono definire anche classi i cui elementi non sono oggetti.

```
let rec
Persone class
   Persona <−>
      [ NomeECognome  :string ;
        Codice        :string ;
        Presentati    := meth ():string is
                           “Il mio nome e’:” & self.NomeECognome
      ] key (Codice)
```

La definizione di una classe introduce nell’ambiente i seguenti legami:

– l’identificatore Persona legato al nuovo tipo oggetto introdotto;
– l’identificatore Persone legato ad una sequenza modificabile, inizialmente vuota, di elementi di tipo Persona;
– l’identificatore mkPersona legato ad una funzione per creare un nuovo oggetto del tipo Persona e per inserirlo alla testa della classe Persone;
– l’identificatore dropPersona legato ad una funzione per eliminare un oggetto dalla classe Persone e per marcarlo come non valido.

Poiché il nome di una classe denota una sequenza di valori, sulle classi sono disponibili tutti gli operatori che operano su sequenze.

1.10.1 Le associazioni

La rappresentazione di associazioni binarie senza proprietà

Le associazioni fra entità sono modellate per *aggregazione*, ovvero definendo oggetti con attributi che restituiscono valori che sono oggetti di altre classi. La modellazione delle associazioni con il meccanismo dell’aggregazione è una scelta tipica di un modello ad oggetti. In particolare, un’associazione binaria si modella aggiungendo in ciascuna delle due classi un attributo con valore gli oggetti associati nell’altra classe. I due attributi rappresentano la funzione *diretta* e la funzione *inversa* dell’associazione. Poiché ciascuno dei due attributi contiene in realtà tutta l’informazione relativa all’associazione, per evitare ridondanza si memorizza uno solo dei due attributi, mentre l’altro viene calcolato da un metodo.

Ad esempio, l’associazione molti a molti tra autori e libri si può rappresentare aggiungendo un attributo memorizzato AutoriDelLibro ai libri ed un attributo calcolato a partire da questo, LibriScritti, agli autori, come nell’esempio seguente:

```
let rec
Autori class
   Autore <−>
      [ Nome           :string ;
        AnnoDiNascita  :int ;
        LibriScritti   := meth (): seq Libro is Libri where self  isin AutoriDelLibro ]
and
Libri class
   Libro <−>
      [ Codice         :int ;
        Argomenti      :seq string ;
        Titolo         :string ;
```

```
    AutoriDelLibro  :seq Autore ];
```

Sarebbe possibile anche evitare del tutto il metodo che calcola l'inversa, che serve solo a facilitare la scrittura delle interrogazioni, oppure memorizzare sia la diretta che l'inversa. In questo caso però il fatto che i due attributi modellino la diretta e l'inversa della stessa associazione andrebbe garantito dalle operazioni di modifica delle classi.

Sfruttando questo approccio, i tipi degli attributi vengono usati per esprimere le caratteristiche di cardinalità e di modificabilità delle associazioni. Un'associazione modificabile da una classe $\mathcal{A}$ verso una classe $\mathcal{B}$ i cui elementi hanno tipo $\mathcal{T}_B$ è rappresentata come segue:

- se è univoca totale viene rappresentata da un attributo memorizzato di tipo **var** $\mathcal{T}_B$, o da un attributo calcolato (un metodo senza parametri) di tipo $\mathcal{T}_B$;
- se è univoca parziale viene rappresentata da un attributo memorizzato di tipo **var optional** $\mathcal{T}_B$, o da un attributo calcolato di tipo **optional** $\mathcal{T}_B$;[7]
- se è multivalore viene rappresentata da un attributo memorizzato di tipo **var seq** $\mathcal{T}_B$, o da un attributo calcolato di tipo **seq** $\mathcal{T}_B$; se si desidera mantenere un eventuale vincolo di totalità, è necessario eseguire esplicitamente il controllo di tale vincolo nelle operazioni di creazione e di aggiornamento.

Se l'associazione è costante e viene rappresentata da un attributo memorizzato, il rispetto del vincolo viene assicurato eliminando il var dal tipo. Se invece l'associazione viene rappresentata da un metodo, il vincolo di costanza va mantenuto esplicitamente dalle operazioni di creazione e modifica.

La rappresentazione di associazioni binarie con proprietà

Un'associazione con proprietà può essere rappresentata da una classe i cui elementi corrispondono alle istanze dell'associazione, e questa via deve essere privilegiata in tutti quei casi in cui non risulti eccessivamente innaturale. Ad esempio, l'associazione tra libri e utenti di una biblioteca, con attributi DataPrestito, e DataRestituzione si modella molto bene come una classe dei Prestiti, che ha gli attributi dell'associazione.

Un'associazione binaria con un attributo si può modellare aggiungendo alle classi un attributo che restituisce, per ogni oggetto x della classe, un insieme di coppie (y, a), una per ogni istanza (x, y) dell'associazione, dove a è il valore dell'attributo dell'associazione per quella istanza.

Consideriamo, ad esempio, due classi di prodotti e fornitori con un'associazione molti a molti tale che (fornitore, prodotto) appartiene all'associazione se fornitore fornisce il prodotto, e con un attributo prezzo che stabilisce a che prezzo quel fornitore fornisce quel prodotto. L'associazione può essere rappresentata in Galileo 97 come una classe oppure, attraverso il meccanismo dell'aggregazione, come segue.

7. Per ogni tipo $\mathcal{T}$, il tipo **optional** $\mathcal{T}$ è un tipo che può assumere o il valore (|unbound|) o un valore (|bound:=$\mathcal{V}$|), dove $\mathcal{V}$ è un valore di tipo $\mathcal{T}$; rimandiamo al manuale per ulteriori dettagli.

```
let rec
Fornitori class
   Fornitore <−>
      [ Offre        :var seq [Prodotto :Prodotto; Prezzo :int ] ]
and
Prodotti class
   Prodotto <−>
      [ OffertoDa  := meth () :seq [Fornitore :Fornitore; Prezzo :int ] is
                        select   [Fornitore:= forn; Prezzo:= offre.Prezzo]
                        from     forn In Fornitori times* offre In (at forn.Offre)
                        where    offre.Prodotto = self];
```

Il vincolo referenziale

Per un qualunque meccanismo utilizzato per rappresentare un'associazione A tra una classe C_1 ed una classe C_2, il vincolo referenziale specifica che gli oggetti che il meccanismo che rappresenta A associa ad un oggetto della classe C_1 sono:

1. oggetti validi,
2. oggetti appartenenti alla classe C_2,

e analogamente, per ciò che riguarda l'altra direzione dell'associazione, gli oggetti associati dal meccanismo ad un oggetto della classe C_2 devono essere validi ed appartenenti alla classe C_1.

Questo vincolo è in genere rispettato dai sistemi fino a quando qualche oggetto non viene rimosso da una classe. In presenza di cancellazioni possono invece crearsi problemi. Si consideri il seguente esempio:

```
let rec
Studenti class
   Studente <−>
      [ Nome             :string ;
        AnnoDiNascita    :int ;
        EsamiSuperati    := meth (): seq Esame is
                                Esami where self= Candidato;
        CorsoDiLaurea    := meth (): CorsoDiLaurea is
                                get (CorsiDiLaurea where selfisin (at Iscritti)) ]
and
Esami class
   Esame <−>
      [ Corso            :string ;
        Voto             :int ;
        Candidato        :Studente ]
and
CorsiDiLaurea class
   CorsoDiLaurea <−>
      [ Nome             :string ;
        Voto             :int ;
        Iscritti         :var seq Studente ];
```

Il vincolo referenziale specifica ad esempio che l'attributo Candidato associa ad un elemento della classe Esami un elemento valido della classe Studenti (e analogamente per ciò che riguarda gli attributi Iscritti, EsamiSuperati e CorsoDiLaurea). Questo vincolo è solo in parte mantenuto dalla tipizzazione del linguaggio, poiché la dichiarazione Candidato :Studente ci assicura solo che l'attri-

buto Candidato è legato ad un oggetto di tipo Studente, ma non necessariamente ad un oggetto valido. In particolare, dopo che è stata applicata l'operazione dropStudente ad uno studente che ha superato un certo esame x, l'espressione x.Candidato non denota più un elemento della classe Studenti, e neppure uno studente valido, ma denota un valore che risponde ad ogni messaggio sollevando un fallimento "killed".

Questo problema è del tutto generale: ogni volta che, in qualunque linguaggio, si cerca di rimuovere un oggetto y da una classe C_2, e c'è un altro oggetto x nella classe C_1 che è associato ad y da un'associazione A, questa operazione può avere almeno quattro esiti, che esemplifichiamo nel caso in cui C_2 sia Studenti ed A, nella direzione da Esami a Studenti, sia rappresentata dall'attributo Candidato:

1. fallimento: il sistema può far fallire l'operazione; in questo esempio, il sistema farebbe fallire la rimozione di uno studente se questi ha esami associati;
2. rimozione dell'istanza di associazione: il sistema rimuove y dalla classe e la coppia x, y dall'associazione, senza cancellare x; se però esiste un vincolo di totalità sull'associazione da C_1 verso C_2, come nel nostro esempio, questo approccio rischia di violare tale vincolo. Infatti, se quando viene rimosso uno studente si rimuovesse anche il valore del campo Candidato da tutti i suoi esami, questi esami resterebbero senza nessuno studente associato, violando il vincolo di totalità. Questo approccio sarebbe invece ragionevole applicandolo all'associazione Iscritti da un corso di laurea verso i propri studenti. In questo caso, la rimozione di uno studente implicherebbe anche la sua rimozione automatica dall'attributo Iscritti del suo corso di laurea, preservando il vincolo referenziale senza violare nessun altro vincolo;
3. rimozione in cascata dell'oggetto associato: il sistema può cancellare automaticamente anche l'oggetto x; questo approccio è rischioso perché può provocare cancellazioni non volute, e quindi un "effetto valanga" se la cancellazione di x provoca a sua volta altre cancellazioni; nel nostro esempio, la cancellazione di uno studente provocherebbe la cancellazione in cascata di tutti gli esami sostenuti da quest'ultimo;
4. violazione del vincolo referenziale: il sistema può ignorare il problema, per cui un successivo tentativo di trovare gli elementi che A associa ad x può portare o a trovare un oggetto che non sta più nella classe, o a trovare un oggetto non più valido, o a sollevare un fallimento a tempo di esecuzione, o a causare un fallimento di sistema. Come già specificato, questo è l'approccio seguito nel Galileo 97 per la diretta delle associazioni.

La soluzione migliore a questo problema, che sarà esemplificata in seguito con riferimento al linguaggio SQL per sistemi relazionali, consiste nel permettere al programmatore di specificare in modo dichiarativo, per ciascuna delle due direzioni di ciascuna associazione, quale dei primi tre comportamenti adottare.

Il programmatore Galileo 97 deve invece affrontare il problema usando il costrutto **before** drop per specificare il comportamento desiderato. Ad esempio, la gestione del vincolo referenziale tramite fallimento, nel caso di rimozione di uno studente con esami associati, si può specificare come segue (i corsi di laurea si tratterebbero in modo analogo).

```
let rec
Studenti class
   Studente <−>
      [ Nome:string ;
        AnnoDiNascita   :int ;
        EsamiSuperati   := meth (): seq Esame is
                                Esami where self= Candidato ]
      before drop(loStudente)
      if     some Esami with Candidato = loStudente
      do     failwith "Drop fallita: studente con esami"
and
Esami class
   Esame <−>
      [ Corso      :string ;
        Voto       :int ;
        Candidato  :Studente ];
```

La gestione tramite rimozione in cascata si otterrebbe invece con la clausola **before** drop sotto specificata.

```
before drop(loStudente)
if     true
do     select   dropEsame(e)
       from     e In loStudente.EsamiSuperati
```

La gestione tramite rimozione dell'istanza di associazione, possibile solo per l'associazione con i corsi di laurea, si otterrebbe togliendo lo studente dagli iscritti al corso di laurea, con la clausola **before** drop sotto specificata.

```
before drop(loStudente)
if     true
do     loStudente.CorsoDiLaurea.Iscritti <−
            select   s
            from     s In (at loStudente.CorsoDiLaurea.Iscritti)
            where    Not (s = loStudente)
```

Osserviamo che, in Galileo 97 e in ogni altro linguaggio, quando una direzione dell'associazione è rappresentata da un metodo, allora il relativo vincolo referenziale è mantenuto automaticamente con il secondo dei quattro approcci sopra delineati. Con riferimento al metodo EsamiSuperati, se si rimuove un esame dato da uno studente x, un successivo tentativo di accedere agli esami superati da x ritornerà tutti gli esami dati dallo studente tranne quello rimosso, in accordo con l'approccio che abbiamo chiamato "rimozione dell'istanza di associazione". Anche in questo caso, comportamenti diversi si possono ottenere usando la clausola **before** drop.

1.10.2 Sottoclassi

Sia $\mathcal{A}$ una classe, o una sottoclasse, e $\mathcal{T}$ il tipo dei suoi elementi. Una sua sottoclasse $\mathcal{A}'$ con elementi di tipo $\mathcal{T}'$, definito per ereditarietà da $\mathcal{T}$, viene definita con il costrutto:

$\mathcal{A}'$ **subset of** $\mathcal{A}$ **class**
$\mathcal{T}' <-> $ **is** $\mathcal{T}$ **and** $\mathcal{H}$
[**before** mk($\mathcal{X}$) **if** $\mathcal{B}$ **do** $\mathcal{E}$]

[**before** drop($\mathcal{X}$) **if** $\mathcal{B}$ **do** $\mathcal{E}$]
[**key** (ListaCampi)]

Classi e sottoclassi sono legate da due vincoli:

- *vincolo strutturale*: se $\mathcal{A}'$ è sottoclasse di $\mathcal{A}$, allora $\mathcal{T}'$ è definito per eredità da $\mathcal{T}$ e, quindi, il tipo degli elementi di una sottoclasse è sottotipo del tipo degli elementi della superclasse;
- *vincolo estensionale*: se $\mathcal{O}$ è un elemento di $\mathcal{A}'$ e $\mathcal{A}'$ è sottoclasse di $\mathcal{A}$, allora $\mathcal{O}$ è un elemento di $\mathcal{A}$.

Inoltre, se $\mathcal{A}'$ è sottoclasse di $\mathcal{A}$, allora $\mathcal{A}'$ eredita i vincoli di integrità definiti su $\mathcal{A}$ usando i costrutti **before** mk e **key**.

Il vincolo estensionale viene mantenuto automaticamente dal sistema. In particolare, quando si crea un valore $\mathcal{O}$ del tipo $\mathcal{T}'$ degli elementi di una sottoclasse con l'operatore mk$\mathcal{T}'$, esso diventa automaticamente un elemento della sottoclasse corrispondente e di tutte le sue superclassi. Quando ad un valore $\mathcal{O}$ viene rimosso il tipo $\mathcal{T}'$ con la funzione drop$\mathcal{T}'$, esso viene rimosso dalla classe corrispondente e da tutte le sottoclassi (ma rimane nelle altre classi a cui appartiene).

Si noti quindi la differenza fra le nozioni di sottotipo e sottoclassi: la nozione di sottotipo consente di sostituire valori di un tipo a valori di un supertipo; la nozione di sottoclasse consente di stabilire che un insieme di valori è sottoinsieme di un altro.

Ecco un esempio di definizione di sottoclassi:

```
let rec
Studenti subset of Persone class
   Studente <-> is Persona and
      [ Matricola   :int ;
        Presentati  := meth ():string is
                          super.Presentati & ", matricola n. " & stringofint(self.Matricola)
      ] key (Matricola)
and
Atleti subset of Persone class
   Atleta <-> is Persona and
      [ Disciplina   :string ];
```

Con le modalità viste per la definizione di sottoclassi, e con gli operatori visti finora per aggiungere o togliere elementi da classi, con riferimento ai quattro tipi di sottoclassi visti nel Capitolo 2, le classi del Galileo 97 corrispondono alle sottoclassi disgiunte senza vincolo di copertura. Ad esempio, Studenti e Atleti sono sottoclassi disgiunte della classe Persone, ma non ne costituiscono una partizione perché, ad esempio, si può costruire una persona che non sia né studente né atleta. Per imporre il vincolo di copertura bisogna impedire la costruzione di elementi nella classe Persone e l'eliminazione di elementi dalle sottoclassi Studenti e Atleti ponendo:

```
let mkPersona := nil
and dropStudente := nil
and dropAtleta := nil;
```

1.10.3 Classi virtuali

Il linguaggio consente di definire con il costrutto Ide := **derived** Exp identificatori legati a valori che vengono calcolati ogni volta che si usa Ide. Sfruttando questa possibilità si possono definire delle sequenze di valori che non sono memorizzate ma che vengono calcolate ogni volta che si opera su di esse. Queste sequenze sono chiamate *classi virtuali*, o *viste*, per analogia con le *view* dei sistemi relazionali. Le classi virtuali sono utilizzate per dare una diversa prospettiva dei dati alle applicazioni, in particolare per realizzarne l'indipendenza logica.

Ad esempio, a partire dalla base di dati vista in precedenza con le classi Persone, Studenti, ed Atleti si può definire le seguente vista, che contiene un sottoinsieme delle persone.

```
let PersoneNonStudenti := derived
        select   x
        from     x In Persone
        where    Not (x isalso Studente);
```

In una vista, oltre a selezionare un sottoinsieme degli elementi di una classe, si desidera in genere modificarne la struttura, ad esempio togliendo, o aggiungendo alcuni attributi. Questo scopo non può essere raggiunto usando gli operatori sui record, poiché si perderebbe l'identità dell'oggetto di partenza. È quindi necessario aggiungere al linguaggio operazioni per costruire una vista di un oggetto la quale nasconde, ridenomina o aggiunge attributi rispetto a quelli presenti nell'oggetto di partenza pur condividendone l'identità. Come esempio di linguaggio con questi operatori si veda [AAG00].

Con questo meccanismo, la vista può essere interrogata come se fosse una classe, ed ogni modifica effettuata ad un oggetto virtuale della vista si riflette sull'oggetto di partenza. Se si vuole rendere la vista indistinguibile da una vera classe è però necessario definire esplicitamente anche le funzioni mk e drop relative, tipicamente definendole in modo che vadano a modificare la classe di partenza in modo opportuno.

Questo non è il solo approccio possibile alla definizione di classi virtuali. Il sistema O_2, ad esempio, anziché dare un operatore di ambiente d'uso generale, come l'operatore **derived** , ed operazioni per costruire oggetti virtuali, mette direttamente a disposizione un meccanismo per la definizione di classi virtuali, che permette sia la selezione che la ristrutturazione degli oggetti coinvolti.

1.11 I vincoli d'integrità

Per trattare i vincoli d'integrità un linguaggio per basi di dati deve prevedere meccanismi (a) per descriverli nello schema, possibilmente in modo dichiarativo, (b) per stabilire quando vanno controllati e (c) per specificare l'azione da intraprendere qualora vengano violati.

1.11.1 Descrizione dichiarativa dei vincoli

Il Galileo 97 è stato disegnato per poter controllare staticamente un'ampia classe di vincoli con il meccanismo dei tipi di dati. Ad esempio, il meccanismo dei tipi assicura, con un'analisi testuale, che ad un oggetto di un certo tipo si applichino solo le operazioni permesse, oppure che i valori costanti non vengano modificati.

Altri tipi di vincoli d'integrità possono essere espressi nel linguaggio con i trigger definiti nei tipi astratti e nelle classi, e con il trigger predefinito **key**.

Ad esempio, per dichiarare i vincoli che il voto di un esame è compreso fra 18 e 30, con eventuale lode, la classe si definisce come segue:

```
let type TVoto <-> int
   before mk(voto)
   if     Not ((voto >= 18) And (voto <= 30))
   do     failwith "Il voto deve essere fra 18 e 30";

let rec
Esami class
   Esame <->
      [ Candidato  :Studente;
        Materia    :string ;
        Voto       :TVoto;
        Lode       :bool ]
      before mk(e)
      if     (e.Voto < 30 And e.Lode)
      do     failwith "Lode ammessa solo con 30"
      key (Candidato, Materia);
```

I vincoli specificati in questo modo vengono verificati prima di creare l'oggetto relativo.

1.11.2 Immersione dei vincoli nelle procedure

Quando non è possibile esprimere un vincolo con i trigger, ad esempio perché riguarda un attributo modificabile o perché è un vincolo dinamico, si procede nel seguente modo:

- si incapsula la parte di stato da cui dipende tale vincolo, ovvero la si memorizza in attributi privati di uno o più oggetti;
- si inseriscono nel codice dei metodi che modificano tale stato tutti i controlli necessari a prevenire la violazione del vincolo.

Ad esempio, un vincolo dinamico che stabilisce che l'età di una persona può solo crescere, può essere rappresentato come segue:

```
let rec
Persone class
   Persona <->
      [ private miaEta  :var int ;
        eta             := meth () :int is at self.miaEta;
        setEta          := meth (newEta:int ) :null is
                              if newEta < (at self.miaEta)
                              then failwith "L'eta' deve crescere"
                              else self.miaEta <- newEta ];
```

1.11.3 Violazione dei vincoli e fallimenti

I *fallimenti*, o *eccezioni*, sono prodotti nella valutazione dalle operazioni predefinite quando si verifica una condizione di errore. Possono inoltre essere prodotti esplicitamente con il costrutto "**failwith** Nome", usato, come abbiamo visto, nella definizione di vincoli sia attraverso il meccanismo dei trigger che attraverso l'immersione nelle procedure. Un fallimento ha sempre associato un nome che ne identifica la natura, e che nel caso delle operazioni predefinite è uguale al nome dell'operazione. I fallimenti possono essere controllati dal programmatore con due modalità: una generale, per qualsiasi fallimento, l'altra selettiva, in cui vengono specificati i nomi dei fallimenti che si vogliono controllare. Vediamo la prima modalità.

Nell'espressione "E_1 **iffails** E_2", se la valutazione di E_1 fallisce, allora il valore dell'intera espressione è quello di E_2, altrimenti è quello di E_1. E_1 ed E_2 devono essere dello stesso tipo. Diciamo in questo caso che E_2 è il gestore dei fallimenti sollevati da E_1. Si noti che, in Galileo 97, anche se E_1 fallisce, gli effetti di E_1 sui dati persistenti non vengono disfatti, perché il Galileo 97 non supporta un meccanismo di transazioni.

1.12 Il linguaggio di interrogazione

Essendo il Galileo 97 un linguaggio interattivo, le interrogazioni si formulano connettendosi alla base di dati desiderata e scrivendo un'espressione con gli operatori previsti sulle classi e sulle strutture dati del linguaggio. La navigazione tra gli oggetti della base di dati, cioè il passaggio da un oggetto di una classe ad un altro in relazione con esso, è facilitata dal meccanismo dell'aggregazione. Nella Sezione 1.4.2 si trovano alcuni semplici esempi di interrogazioni. Riportiamo qui altri esempi che coinvolgono l'esplorazione di associazioni, facendo riferimento alle classi definite nella Sezione 1.10.1.

Nel primo esempio mostriamo come sia possibile esaminare tutti i libri scritti da ogni autore generando tutte le possibili coppie libro-autore, e poi restringendosi a quelle in cui l'autore è uno degli autori del libro. L'interrogazione ritorna il nome dell'autore e il titolo del libro per tutti i testi che trattano anche di informatica:

```
select   [ Aut := A.Nome; Lib := L.Titolo ]
from     L In Libri times* A In Autori
where    "Informatica" isin L.Argomenti And A isin L.AutoriDelLibro;
```

Lo stesso effetto può essere ottenuto utilizzando un prodotto dipendente. Ricordando che `L In Libri times* A In L.AutoriDelLibro` ritorna tutte le coppie formate da un libro `L` e da un autore `A` tratto dal campo `AutoriDelLibro` di `L`, riscriviamo l'interrogazione come segue:

```
select   [ Aut := A.Nome; Lib := L.Titolo ]
from     L In Libri times* A In L.AutoriDelLibro
where    "Informatica" isin L.Argomenti;
```

Lo stesso risultato potrebbe essere ottenuto accoppiando ogni autore con i suoi libri, ovvero sostituendo `L In Libri times* A In L.AutoriDelLibro` con `A In Autori`

times* L **In** LibriScritti. Gli identificatori L ed A non sono necessari, anche se aumentano la leggibilità dell'interrogazione, che potrebbe anche essere scritta come segue:

```
select   [ Aut := Nome; Lib := Titolo ]
from     Libri times* AutoriDelLibro
where    "Informatica" isin Argomenti;
```

Il prossimo esempio illustra l'uso di quantificatori per esprimere condizioni più complesse. L'interrogazione riporta autore e titolo per quei testi tali che: (a) il testo parla *solo* di informatica e (b) l'autore del testo ha scritto solo testi che trattano *anche* di informatica; gli identificatori A, L ed L2 sono stati usati solo per aumentare la leggibilità:

```
select   [ Aut := A.Nome; Lib := L.Titolo ]
from     A In Autori times* L In A.LibriScritti
where    L.Argomenti = "Informatica"
         And (each L2 In A.LibriScritti with ("Informatica" isin L2.Argomenti));
```

Il prossimo esempio esegue la stessa interrogazione associando però le condizioni ai fattori anziché al risultato del prodotto dipendente:

```
select   [ Aut := Nome; Lib := Titolo ]
from     (Autori where each LibriScritti with "Informatica" isin Argomenti)
           times*
         (LibriScritti where Argomenti = {"Informatica"});
```

1.13 Riepilogo

Nelle sezioni precedenti si è mostrato come trattare i seguenti aspetti del modello ad oggetti:

- definizione e costruzione di oggetti con identità, stato e comportamento;
- definizioni per ereditarietà stretta di tipi oggetto;
- definizione di collezioni di oggetti organizzate in una gerarchia di sottoinsieme con il meccanismo delle classi;
- modellazione di associazioni fra collezioni di oggetti con il meccanismo dell'aggregazione.

Queste possibilità sono offerte da quasi tutti i linguaggi ad oggetti per basi di dati, anche se essi non prevedono necessariamente un controllo statico e forte dei tipi.

Nella prossima sezione si mostra un'importante funzionalità che non è di solito offerta dagli attuali sistemi commerciali: la modifica dinamica del tipo degli oggetti. In seguito si discutono alcune scelte alternative a quelle effettuate nel linguaggio Galileo 97. Infine, si mostrano le soluzioni offerte da due altri linguaggi di sistemi a oggetti per mostrare analogie e differenze da quanto visto col Galileo 97.

1.14 Oggetti che cambiano tipo

Chiamiamo *estensione* di un oggetto l'operazione con la quale un oggetto che ha un certo tipo (ad esempio, Persona) può acquisire anche uno o più sottotipi di tale tipo (ad esempio, Studente), senza cambiare l'identità. L'operazione di estensione è molto importante per modellare situazioni comuni nel mondo reale (ad esempio, una persona che diventa uno studente), ma non è in genere offerta dai linguaggi per basi di dati ad oggetti.

Si dice che un oggetto ha *pluralità di comportamenti* quando un singolo oggetto ha a disposizione più *ruoli*, e la risposta che dà ad un messaggio dipende dal ruolo con cui viene osservato. La *pluralità di comportamenti* è estremamente utile quando viene offerta l'operazione di estensione, per motivi che vengono ora illustrati.

Nei linguaggi senza estensione, ogni oggetto ammette un *tipo minimo*, ovvero un tipo $\mathcal{T}$ tale che tutti gli altri tipi a cui l'oggetto appartiene sono supertipi di $\mathcal{T}$; questo tipo minimo è il tipo con cui l'oggetto è stato creato. L'esistenza di un tipo minimo garantisce che, per ogni metodo che è definito più volte nella gerarchia, esiste per ogni oggetto una versione più specializzata di quel metodo che l'oggetto userà per rispondere al corrispondente messaggio. Nei linguaggi con estensione non tutti gli oggetti hanno un tipo minimo. Ad esempio, se si estende un oggetto luigi di tipo Studente facendovi assumere anche il tipo Atleta, sottotipo di Persona, luigi non avrà più un tipo minimo, ma solo due tipi minimali, Studente ed Atleta. Se Studente ed Atleta definiscono entrambi un campo o un metodo con lo stesso nome, non ci si può aspettare che le due definizioni siano una specializzazione l'una dell'altra; si immagini, ad esempio, che i due tipi definiscano entrambi un campo Matricola, dove una è la matricola dello studente e l'altra identifica il tesserato ad un'associazione sportiva. In questa situazione, è necessario modificare il meccanismo dell'interpretazione dei messaggi (e dell'accesso ai campi, che può esserne considerato un caso particolare) per far sì che un oggetto come luigi sia sempre interrogato o nel suo ruolo di studente o in quello di atleta, e risponda al messaggio Matricola in due modi diversi nelle due situazioni.

Consideriamo ora la situazione di un oggetto O di tipo Persona, che ha già un metodo per rispondere al messaggio Presentati, e che acquisisce il tipo Studente che ridefinisce tale metodo. Avendo pluralità di comportamenti, O potrebbe continuare ad utilizzare il vecchio metodo quando è interrogato come persona, ed utilizzare il nuovo quando è acceduto come studente. Però, dato che Studente è un sottotipo di Persona, il comportamento del nuovo messaggio Presentati dovrebbe essere una specializzazione del comportamento del vecchio messaggio, per cui si potrebbe anche stabilire che O utilizzerà il nuovo metodo anche quando è interrogato nel suo ruolo di Persona. La soluzione più ricca è quella di permettere al programmatore di specificare, quando manda un messaggio ad un oggetto attraverso un certo ruolo, se desidera che l'oggetto usi il metodo proprio di tale ruolo oppure se cercare il metodo tra le eventuali specializzazioni di tale ruolo.

Un meccanismo per trattare oggetti con estensione e con pluralità di comportamenti è stato proposto e realizzato per i linguaggi Fibonacci e Galileo 97, entrambi con controllo statico dei tipi [ABGO93].

Seguendo l'approccio del Galileo 97, quando un tipo oggetto T′ è definito come sottotipo del tipo T, viene definita sia la funzione mkT′ per costruire direttamente nuovi valori di tipo T′, sia la funzione inT′ per estendere un valore di tipo T in uno di tipo T′, senza alterarne l'identità.[8] La funzione inT′ ha due parametri: il valore dell'oggetto O da estendere e un record che specifica i valori dei campi di T′ non ereditati da T.

Riprendiamo l'esempio del tipo oggetto Persona visto in precedenza:

```
let rec
type Persona <−>
        [ Codice        :string ;
          Nome          :string ;
          AnnoNascita   :int ;
          Telefono      : [ Abitazione : var string ];
          Presentati    := meth ():string is "Mi chiamo " & self.Nome ]
        before mk(p)
        if      p.AnnoNascita < 1900
        do      failwith "AnnoNascita deve essere >= 1900" ;

let Mario := mkPersona (
        [ Nome          := "Mario Rossi";
          Codice        := "rssmar23h67";
          AnnoNascita   := 1967;
          Telefono      := [ Abitazione := var "06 222444" ] ]);
```

L'oggetto Mario può essere esteso con il tipo Studente come segue:

```
let rec
type Studente <−> is Persona and
        [ Matricola         :string ;
          CorsoDiLaurea     :string ;
          Telefono          :[ Abitazione :var string ; Dimora :var string ];
          Presentati        := meth ():string is
                                  super.Presentati &
                                  " Sono iscritto al corso di laurea " &
                                  self.CorsoDiLaurea ];

let StudenteMario :=
        inStudente(Mario, [ Matricola := "A123";
                            CorsoDiLaurea := "Informatica";
                            Telefono := [ Abitazione := Mario.Telefono.Abitazione;
                                          Dimora := var "552244" ] ]);
```

Supponiamo che venga definito anche il tipo Atleta e poi si estenda Mario nuovamente con questo tipo:

```
let rec
type Atleta <−> is Persona and
        [ Matricola:int ;
          Sport         : string ;
          Presentati    := meth ():string is
                              super.Presentati &
```

8. Se T è il tipo degli elementi di una classe A e T′ il tipo degli elementi di una sua sottoclasse B, quando un elemento di A di tipo T viene esteso con il tipo T′ esso diventa anche un elemento della sottoclasse B.

```
                    " Sono un professionista di " &
                    self.Sport ];

let AtletaMario := inAtleta(Mario, [ Matricola := 245; Sport := "tennis" ]);
```

Mario, StudenteMario ed AtletaMario denotano tre *ruoli* diversi dello stesso oggetto. Mario denota l'oggetto nel suo ruolo di Persona, mentre StudenteMario ed AtletaMario lo denotano nei suoi ruoli di Studente e di Atleta, per cui mandando il messaggio Matricola a StudenteMario e ad AtletaMario otterremo, rispettivamente, "A123" e 245, ed analogamente mandando ai due ruoli il messaggio Presentati si otterranno due risposte diverse.

Per quanto riguarda il ruolo Mario, non è possibile mandargli un messaggio Matricola, perché tale campo non è previsto per i valori di tipo Persona. Per ciò che riguarda il messaggio Presentati, il Galileo 97 prevede due diversi modi di mandarlo:

- ricerca doppia: utilizzando la notazione Mario.Presentati si chiede che il metodo sia cercato prima nei sottoruoli del ruolo corrente, a partire dall'ultimo sottoruolo acquisito, poi nel ruolo corrente e infine nei ruoli antenati; Mario risponderà quindi usando il suo metodo come Atleta;
- ricerca verso l'alto: utilizzando la notazione Mario!Presentati si chiede che il metodo sia cercato, ignorando i sottoruoli, a partire dal ruolo corrente e poi ruoli antenati; Mario risponderà quindi usando il suo metodo come Persona.

Quindi, la risposta al messaggio Presentati inviato a Mario con la modalità della ricerca doppia cambia una prima volta dopo la sua estensione con il tipo Studente e una seconda volta dopo la sua estensione con il tipo Atleta, mentre con la ricerca verso l'alto la risposta non cambia, indipendentemente dalle estensioni a cui viene sottoposto.

È importante osservare che il ruolo denotato da un'espressione non dipende dal tipo assegnato dal compilatore all'espressione, ma solo dal valore denotato dall'espressione stessa. Ad esempio, nella seguente espressione, M è un valore a cui il compilatore assegna il tipo Persona, tuttavia le tre iterazioni della clausola select accedono a tre ruoli diversi dell'oggetto Mario; anche l'esempio successivo illustra lo stesso punto.

```
let Marii := {AtletaMario; StudenteMario; Mario} :seq Persona;

select   M!Presentati
from     M In Marii;

> {"Mi chiamo Mario Rossi. Sono un professionista di tennis.";
>  "Mi chiamo Mario Rossi. Sono iscritto al corso di laurea Informatica.";
>  "Mi chiamo Mario Rossi."} :seq string

let AtletaMarioDiTipoPersona := AtletaMario :Persona;

AtletaMarioDiTipoPersona!Presentati;

> "Mi chiamo Mario Rossi. Sono un professionista di tennis.":string
```

Altri operatori su oggetti sono:

– dropT(Expr), per rimuovere il tipo T e i suoi sottotipi posseduti dall'oggetto denotato dall'espressione Expr, e per rimuovere l'oggetto dalle classi eventualmente associate a questi tipi. Dopo avere eseguito questa operazione, ogni tentativo di accedere all'oggetto attraverso uno dei ruoli che corrispondono ai tipi fallisce. Ad esempio, dopo aver eseguito dropAtleta(AtletaMario), l'espressione AtletaMario.Sport genera un fallimento.
– Expr **isalso** T, per controllare se l'oggetto denotato dall'espressione Expr ha anche il ruolo di tipo T; ad esempio Mario **isalso** Atleta è vero, come pure StudenteMario **isalso** Atleta.
– Expr **As** T, per restituire l'oggetto O denotato dall'espressione Expr con il ruolo di tipo T, che deve essere uno dei tipi posseduti da O; ad esempio AtletaMario **As** Studente restituisce il ruolo di tipo Studente posseduto dall'oggetto con ruolo AtletaMario.
– Expr **isexactly** T per controllare se il ruolo denotato dall'espressione Expr è di tipo T; ad esempio Mario **isexactly** Atleta è falso, mentre AtletaMario **isexactly** Atleta è vero.

1.14.1 Estensione e tipi di sottoclassi

Grazie all'operazione di estensione, le sottoclassi $\mathcal{A}_1 \ldots \mathcal{A}_n$ di una classe $\mathcal{A}$, con elementi di tipo $\mathcal{T}_1 \ldots \mathcal{T}_n$, non sono insiemi disgiunti; infatti, un elemento di $\mathcal{A}$ può diventare un elemento delle sottoclassi $\mathcal{A}_1 \ldots \mathcal{A}_n$ estendendolo con gli operatori in$\mathcal{T}_1 \ldots$ in$\mathcal{T}_n$; l'aggiunta dell'operatore di estensione rimuove quindi il vincolo di disgiunzione.

Abbiamo già visto nella Sezione 1.10.2 come il vincolo di copertura si possa imporre ridefinendo alcune delle operazioni mk, drop. In modo analogo, il vincolo di disgiunzione tra più sottoclassi legate ai tipi A_1 e A_n si può imporre ridefinendo ciascuna operazione inA_i in modo che non abbia effetto, oppure in modo che sollevi un fallimento se l'oggetto appartiene già ad una sottoclasse legata ad uno dei tipi A_j.

1.15 Alcune scelte alternative

1.15.1 Tipi oggetto con implementazione multipla

Nei sistemi che supportano i tipi oggetto con implementazione multipla, la definizione di un tipo oggetto ne specifica solo l'interfaccia ma non l'implementazione.

Ad esempio, immaginando di avere a disposizione una variante del Galileo 97 (che in realtà non esiste) che supporta l'approccio con implementazione multipla, la definizione di due tipi Persona e Studente ne conterrebbe solo l'interfaccia e potrebbe essere scritta come segue:

```
let type Persona <->
        [ Codice        :string ;
          Nome          :string ;
          Telefono      :int ;
          setTelefono   :int -> null ;
          Presentati    :string ];
```

```
let type Studente <−> is Persona and
            [ Matricola        :string ;
              CorsoDiLaurea    :string ;
              Presentati       :string ];
```

La struttura dello stato e la realizzazione dei metodi vengono invece specificati definendo un costruttore, cioè una funzione che costruisce oggetti specificandone l'implementazione.

Il costruttore può costruire l'oggetto da zero, utilizzando l'operatore mk$\mathcal{T}$, a cui in questa variante del linguaggio va specificata l'intera implementazione dell'oggetto, che deve rispettare l'interfaccia associata al tipo $\mathcal{T}$. Alternativamente, il costruttore può costruire l'oggetto per ereditarietà, ovvero chiamando un altro costruttore ed estendendo il risultato di tale costruzione usando una variante dell'operatore in$\mathcal{T}$, come avviene nel costruttore creaStudente sottostante.

```
let creaPersona :=
   fun(UnCodice:string , UnNome:string , UnTel:int ): Persona
      is mkPersona(
         [ private MioTelefono := var UnTel;
           Codice := UnCodice;
           Nome := UnNome;
           Telefono := meth () :int is (at self.MioTelefono);
           setTelefono:= meth (newTel:int ): null is self.MioTelefono <− newTel;
           Presentati := meth ():string is "Mi chiamo " & self.Nome ]);

let creaStudente :=
   fun(UnCod:string , UnNome:string , UnTel:int UnCDL:string , UnaMat:string ): Persona is
    inStudente(creaPersona(UnCod,UnNome,UnTel),
       [ CorsoDiLaurea := UnCDL;
         Matricola := UnaMat;
         Presentati := meth ():string is
               super.Presentati &
                  ", e sono studenti di " &
                  self.CorsoDiLaurea ] );
```

In questo approccio è possibile definire costruttori diversi per uno stesso tipo oggetto, ed è anche possibile usare gli operatori mk ed in per definire un singolo oggetto, che avrà quindi un'implementazione diversa da qualunque altro oggetto presente nel sistema.

Da un confronto tra questo approccio e l'approccio ad implementazione singola del Galileo 97, si possono trarre le seguenti conclusioni.

Il modello dei tipi oggetto basato sull'implementazione singola comporta che la definizione di un tipo contenga dentro di sé anche la struttura dello stato e l'implementazione dei metodi. Ne segue che la semantica di un tipo oggetto risulta una cosa molto complessa, e da questo consegue che il linguaggio, sotto l'apparente semplicità, presenta una certa complessità semantica nascosta.

Nel modello ad implementazioni multiple, viceversa, un tipo oggetto definisce solo un'interfaccia ed un insieme di supertipi. Questa maggiore semplicità è estremamente importante quando il livello dei tipi viene arricchito con costrutti quali moduli e polimorfismo. Inoltre, il modello ad implementazioni multiple si presta bene ad applicazioni in cui si integrano oggetti che provengono da sorgenti diverse, dei quali si conosce l'interfaccia ma non l'imple-

mentazione. Infine, questo modello si presta meglio a gestire situazioni in cui si rende opportuno, in una fase della vita di un sistema, modificare l'implementazione degli oggetti che verranno creati in futuro, ma non si vorrebbe dover modificare anche la realizzazione di tutti gli oggetti già creati in precedenza. Un vantaggio invece dell'approccio ad implementazione singola riguarda l'ottimizzazione del linguaggio. Con questo approccio, infatti, il compilatore può dedurre alcune informazioni relativamente alla struttura di un oggetto a partire dal suo tipo e può sfruttare queste informazioni per effettuare importanti ottimizzazioni, anche se la presenza della gerarchia di sottotipo riduce in qualche misura questa possibilità.

1.15.2 Classi con inserzione esplicita

In un linguaggio che supporta il meccanismo delle classi con inserzione esplicita, la definizione di una classe è indipendente dalla definizione di un tipo oggetto, ed in ogni momento è possibile definire una nuova classe specificando il tipo dei suoi elementi ed eventuali vincoli di inclusione o di disgiunzione con altre classi. Essendo classi e tipi oggetto disgiunti, la creazione di un oggetto non lo inserisce automaticamente in alcuna classe e la rimozione di un oggetto da una classe non modifica la validità dell'oggetto.

Ad esempio, il seguente codice, scritto in un'altra variante (immaginaria) del Galileo 97 che supporti le classi con inserzione esplicita, crea una classe Persone di persone, una sua sottoclasse Studenti e un'altra classe Amici di persone, facendo riferimento ad un tipo Persona e ad un sottotipo Studente che sono stati definiti in precedenza. Crea poi uno studente luigi (usando un costruttore creaStudente che supponiamo essere già stato definito), lo inserisce nella classe degli studenti, e più tardi lo rimuove da tale classe, senza però che l'oggetto perda la sua validità.

```
let Persone := emptyClass of Persona;
let Studenti := emptyClass of Studenti are Persone;
let Amici := emptyClass of Persona;

let luigi := creaStudente("fgdlgu25g15b157h", "Luigi", 405067, "Chimica", "A123");
insert luigi into Studenti;
...
remove luigi from Studenti;
```

La scelta tra inserzione automatica, con al più una classe per ogni tipo oggetto, ed inserzione esplicita, con più classi per ogni tipo oggetto, è relativamente meno importante rispetto a quella discussa nel capitolo precedente. Anche in questo caso, nell'approccio ad inserzione automatica, il fatto di legare un aspetto del linguaggio dei valori (la classe) ad un aspetto del linguaggio dei tipi complica la semantica del linguaggio. Un riflesso di questa complessità si coglie nella gestione del vincolo referenziale: nel modello ad inserzione automatica si approssima il vincolo di appartenenza di un oggetto ad una classe con il vincolo di appartenenza al tipo associato, e questa approssimazione crea dei problemi in presenza dell'operazione di rimozione, come visto nella Sezione 1.10.1. Il modello ad inserzione esplicita permette invece di separare la

rimozione da una classe dall'operazione di cancellazione di un oggetto, e di trattare le relative problematiche in maniera indipendente.

In conclusione, l'approccio dell'implementazione singola e dell'inserzione automatica permette di definire classi e tipi oggetto in maniera molto compatta e di iniziare rapidamente ad operarvi, ed è stato adottato per questo motivo nel linguaggio didattico Galileo 97. L'approccio con implementazioni multiple ed inserzione esplicita richiede invece di scrivere maggior codice per definire esempi anche piccoli, ma permette di mantenere una netta separazione tra il livello dei valori e quello dei tipi a vantaggio della semplicità intrinseca del linguaggio.

1.15.3 La rappresentazione esplicita dell'associazione

Come si è visto in precedenza, nel modello ad oggetti le associazioni fra entità sono modellate per *aggregazione* e manca un meccanismo per modellarle separatamente, come accade nel modello entità-relazioni. In alcuni linguaggi è previsto un tale meccanismo, che rappresenta un'associazione come un insieme di record, al fine di superare alcuni limiti dell'aggregazione nella modellazione di associazioni. Una delle proposte più complete è quella presente nel linguaggio Fibonacci [AGO95], un linguaggio per basi di dati ad oggetti definito e realizzato presso l'Università di Pisa, caratterizzato tra l'altro dall'adozione del modello dei tipi oggetto con implementazione multipla e delle classi con inserzione esplicita. La rappresentazione esplicita delle associazioni permette di specificare per ogni associazione non solo i vincoli di molteplicità e di totalità su tutte le componenti, ma anche con quale tecnica va mantenuto ciascun vincolo referenziale, scegliendo tra fallimento, rimozione dell'istanza di associazione, rimozione dell'oggetto dalla classe (Sezione 1.10.1). Inoltre, la rappresentazione di associazioni non binarie o con attributi è immediata.

Supponiamo, ad esempio, che LeAuto e Persone siano due classi. La seguente associazione tra auto e persone è univoca per le auto (**key** (Auto)), totale sulle auto (**onto** Auto), e il vincolo referenziale va mantenuto con la tecnica del fallimento su entrambe le classi (**in** Persone, **in** LeAuto).

```
let proprieta := Assoc of
        [ Proprietario :Persona in Persone;
          Auto         :Auto in LeAuto onto LeAuto ]
        key (Auto);
```

In questo approccio, l'inserimento di una coppia auto-proprietario nell'associazione è un'operazione distinta dalla creazione dell'auto, ma, per non violare il vincolo di totalità, le due operazioni devono essere eseguite contemporaneamente, o almeno all'interno di una singola transazione.

Questo approccio è più espressivo rispetto a quello tradizionale, in particolare per ciò che riguarda vincoli, associazioni non binarie ed associazioni con attributi, ma, a causa di questa ricchezza, è anche più pesante da usarsi.

1.16 I linguaggi dei sistemi O_2 e UniSQL

In questa sezione viene presentato un breve confronto delle caratteristiche del Galileo 97 con quelle dei linguaggi di O_2 e UniSQL per modellare i meccanismi d'astrazione del modello ad oggetti minimo. Faremo riferimento alla base di dati della Figura 1.1, e nelle definizioni degli schemi ad oggetti nei tre linguaggi per brevità si rappresentano solo le dirette delle associazioni. In Figura 1.2 è mostrata la soluzione in Galileo 97.

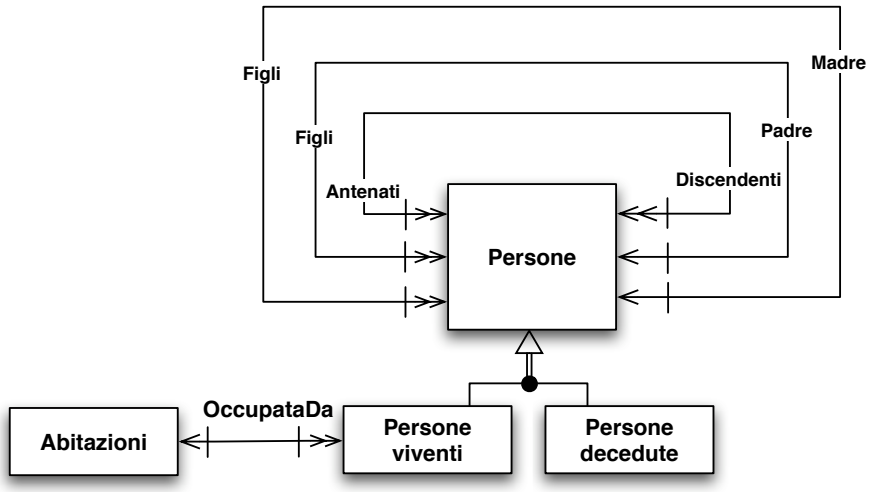

Figura 1.1. Schema di riferimento

1.16.1 Definizione di basi di dati in O_2

O_2, un sistema prodotto dalla società francese O_2*Technology*, adotta l'approccio di immergere nel linguaggio *C* opportuni costrutti per basi di dati ad oggetti. Il linguaggio esteso viene chiamato O_2C.

O_2 tratta *valori* ed *oggetti* con identità. I valori sono descritti da un tipo e gli oggetti sono descritti da un tipo oggetto (che viene chiamato "classe"). Gli attributi di un oggetto possono essere di tipo primitivo, oggetto o definito a partire dai precedenti con i costruttori di tipo record, insieme, multiinsieme o sequenza.

Nella definizione di un tipo oggetto, la parola chiave type precede la definizione del tipo rappresentazione e method precede la dichiarazione delle operazioni associate al tipo. La clausola public è usata per dichiarare definizioni accessibili dall'esterno.

Il linguaggio segue l'approccio dell'implementazione singola per i tipi oggetto. Tuttavia, il codice di un metodo non è dato assieme alla definizione del tipo oggetto, ma viene definito in seguito con un comando distinto:

method body <methodSpec> **in class** <className>
{<O2C instructions>}

```
let type Indirizzo := [ Via:string ; Citta:string ];

let rec
Persone class
    Persona <−>
        [ AnnoNascita  :int ;
          Nome         :string ;
          Eta          := meth ():int is AnnoCorrente − self.AnnoNascita;
          Padre        :optional Persona;
          Madre        :optional Persona;
          Antenati     := meth () :seq Persona is
                              (if self.Madre is unbound
                                  then {}: seq Persona
                                  else ((self.Madre as bound)::(self.Madre as bound).Antenati))
                              append
                              (if self.Padre is unbound
                                  then {}: seq Persona
                                  else ((self.Padre as bound)::(self.Padre as bound).Antenati))
        ]
and
PersoneViventi subset of Persone class
    PersonaVivente <−> is Persona and
        [ Dimora       :var Abitazione ]
and
PersoneDecedute subset of Persone class
    PersonaDeceduta <−> is Persona and
        [ AnnoDecesso:int ]
and
Abitazioni class
    Abitazione <−>
        [ Categoria    :string ;
          Indirizzo    :Indirizzo
        ];
```

Figura 1.2. Un esempio in Galileo 97.

Un tipo oggetto può essere definito per ereditarietà singola o multipla con la clausola inherit. È ammessa la ridefinizione degli attributi ereditati con sottotipi e la ridefinizione di un metodo con la regola della *covarianza*.

Il sistema segue l'approccio delle classi con inserzione esplicita, per cui le classi non sono definite automaticamente ma vanno definite esplicitamente come collezioni modificabili di oggetti, e gli oggetti vanno inseriti esplicitamente. Il sistema non gestisce sottoclassi: se il programmatore desidera che PersoneViventi sia inclusa in Persone deve definire la funzione usata per inserire oggetti in PersoneViventi in modo che inserisca l'argomento anche nella collezione Persone.

1.16.2 Definizione di basi di dati in UniSQL

UniSQL è un sistema relazionale ad oggetti prodotto dalla UniSQL Inc. La società è stata fondata nel maggio 1990 da Won Kim, che aveva progettato in precedenza il sistema ad oggetti ORION, che estende Common Lisp con costrutti per basi di dati ad oggetti. UniSQL è molto simile ad ORION ma ha una sintassi tipo quella dei sistemi relazionali.

Una base di dati si definisce con il linguaggio SQL/X, un dialetto dell'SQL-92 esteso con nuovi comandi per definire oggetti, trigger, viste e modifiche dello schema.

Gli attributi di un oggetto possono essere solo di tipo primitivo, di tipo oggetto oppure di tipo insieme, multiinsieme o sequenza. Pertanto, non si può definire un attributo di tipo record e il tipo Indirizzo va definito come ogni altro oggetto. Esiste un unico costruttore di tipo, chiamato *class*, per definire tipi oggetti. Di un metodo si specifica la segnatura e l'archivio in cui si trova la sua definizione data con un linguaggio esterno a UniSQL/X, ad esempio il C. Il nome del tipo T denota anche l'insieme di tutti i valori di quel tipo costruiti con il comando INSERT INTO T ...

Un tipo oggetto può esser definito per ereditarietà singola, o multipla, ed è consentita la ridefinizione degli attributi ereditati e metodi con tipi compatibili.

1.16.3 Confronto

Galileo 97, O_2 e UniSQL offrono linguaggi con soluzioni diverse per modellare basi di dati ad oggetti. Vediamo le differenze più significative.

Oggetti

La nozione di oggetto con identità è prevista nei tre linguaggi. L'incapsulazione dello stato, in modo che esso sia accessibile solo attraverso i metodi associati all'oggetto, è prevista in O_2 e in Galileo 97 con gli attributi privati. In UniSQL gli oggetti sono gli unici valori che possono persistere, mentre in Galileo 97 e O_2 la persistenza è una proprietà ortogonale al sistema dei tipi e quindi ogni valore legato ad un identificatore nell'ambiente globale diventa persistente.

```
typedef Indirizzo := tuple(Citta:string, Via:string)

class Abitazione
public type tuple
    ( Categoria     :string,
      Indirizzo     :Indirizzo)
method
    public Init :Abitazione,
    public Delete;
end


class Persona
public type tuple
    ( Nome          :string,
      AnnoNascita   :integer,
      Padre         :Persona,
      Madre         :Persona)
method
    public Init :Persona,
    public Delete,
    public Eta :integer,
    public Antenati :set(Persona);
end

class PersonaVivente inherit Persona
public type tuple
    ( Dimora        :Abitazione)
method
    public Init :PersonaVivente,
    public Delete;
end

class PersonaDeceduta inherit Persona
public type tuple
    ( AnnoDecesso:integer)
method
    public Init :PersonaDeceduta;
end

name Persone            :set(Persona);
name Abitazioni         :set(Abitazione);
name PersoneViventi     :set(PersonaVivente);
name PersoneDecedute :set(PersonaDeceduta);
```

Figura 1.3. Un esempio in O_2.

```
CREATE CLASS Indirizzo
    ( Via              string,
      Citta            string );

CREATE CLASS Abitazione
    ( Categoria        string,
      Indirizzo        Indirizzo );

CREATE CLASS Persona
    ( AnnoNascita      INTEGER;
      Nome             STRING;
      Padre            Persona;
      Madre            Persona )
      METHOD           Eta () integer,
                       Antenati () set Persona,
      FILE 'demo.o';

CREATE CLASS PersonaVivente
    AS SUBSET OF Persona
    ( Dimora           Abitazione );

CREATE CLASS PersonaDeceduta
    AS SUBSET OF Persona
    ( AnnoDecesso  integer );
```

Figura 1.4. Un esempio in UniSQL.

Tipi di oggetti

Tutti e tre i linguaggi si basano sul modello ad implementazione singola, in cui la definizione di un tipo oggetto specifica sia la struttura dello stato che i metodi associati. Per quanto riguarda la definizione delle proprietà, solo UniSQL impone la restrizione che i valori di una proprietà siano elementari, di tipo oggetto o insieme di oggetti. Negli altri linguaggi esiste un insieme di costruttori di tipo (e.g. *union, record, array, list*) che possono essere utilizzati senza restrizioni nella definizione del tipo del valore di una proprietà. Solo UniSQL impone la restrizione che nello schema di una base di dati si possono definire solo tipi oggetto. In UniSQL è prevista inoltre la definizione di valori di proprietà per *default*.

Solo in Galileo 97 si distingue fra il tipo di una proprietà costante o modificabile, con il vantaggio, fra gli altri, di rendere possibile il controllo di eventuali tentativi di modifica del valore di proprietà costanti al momento della compilazione. In O_2 questa possibilità non rientra nel sistema dei tipi, ma nel meccanismo per definire regole di accesso, che consente di escludere l'operazione di modifica su certe proprietà. In O_2 e UniSQL tutte le proprietà sono modificabili e possono non essere inizializzate al momento della creazione di un oggetto. In Galileo 97 è possibile associare al tipo di un oggetto dei trigger, che in UniSQL sono associati a collezioni di oggetti.

Per quanto riguarda la definizione dei metodi, in Galileo 97 essi sono dati nella definizione del tipo dell'oggetto, e nel corpo di un metodo si può accedere allo stato dell'oggetto. Negli altri linguaggi, nella definizione del tipo oggetto viene dato solo il tipo dei metodi, mentre il corpo viene dato separa-

tamente con una procedura nel linguaggio del sistema, in cui l'oggetto destinatario del messaggio è il primo parametro. Anche se la definizione dell'interfaccia del tipo e del corpo dei metodi sono date in due momenti diversi, O_2 e UniSQL realizzano ugualmente il modello ad implementazione singola, poiché l'implementazione dei metodi è associata al tipo oggetto e non al singolo oggetto.

Per ciò che riguarda la cancellazione degli oggetti, O_2 non prevede un operatore per la cancellazione esplicita, ma lascia al sistema il compito di eliminare automaticamente oggetti che non sono raggiungibili da valori persistenti. Viene però consentito di rimuovere oggetti dalle classi, come descritto più avanti. Galileo 97 prevede un operatore per far perdere un tipo ruolo ad un oggetto e rimuoverlo contemporaneamente da una classe.

Infine, solo il linguaggio Galileo 97 permette un controllo statico e forte dei tipi, ovvero permette la verifica di tutti gli errori di tipo da parte del compilatore, almeno fino a che non si faccia uso dell'operatore di rimozione.

Classi

Galileo 97 e UniSQL si basano entrambi sul modello delle classi ad inserzione automatica. In UniSQL per ogni tipo oggetto viene definita automaticamente una classe con lo stesso nome, che contiene tutti gli oggetti persistenti di quel tipo, mentre in Galileo 97 la classe è definita solo su richiesta del programmatore. In O_2 sono invece implementate le classi con inserzione esplicita, che sono semplicemente valori persistenti modificabili di tipo "insieme".

Le associazioni fra i dati sono descritte in tutte le proposte con il meccanismo dell'aggregazione, ovvero con proprietà che assumono come valori elementi di altre classi.

Sulle classi sono previsti operatori per rimuovere elementi.

Definizioni per ereditarietà e sottotipi

In tutte le proposte è possibile definire tipi oggetto per ereditarietà. Solo in Galileo 97 le regole di ridefinizione delle proprietà garantiscono con un controllo statico che i valori del nuovo tipo siano un sottotipo del tipo più generale. UniSQL e O_2 prevedono l'ereditarietà multipla per i tipi oggetto.

Gerarchie fra classi

La gerarchia di sottoinsieme fra classi è trattata diversamente nelle varie proposte. O_2 non offre questo meccanismo. Galileo 97 offre un meccanismo di sottoclassi, e le sottoclassi di una classe non sono necessariamente disgiunte. In UniSQL le sottoclassi sono disgiunte, tranne in presenza di gerarchie multiple, e si può accedere ad una classe senza "vedere" gli elementi che appartengono anche ad una sottoclasse. Solo in Galileo 97 è previsto un operatore per spostare elementi da una classe ad una sottoclasse ed è possibile definire tutti i quattro tipi di sottoclassi visti nel Capitolo 2.

1.17 Lo standard ODMG

La presenza di diversi sistemi ad oggetti sul mercato, e la necessità di competere con i sistemi relazionali per i quali esiste un linguaggio standard (SQL) per la definizione e l'uso dei dati, ha portato i principali costruttori di sistemi per basi di dati ad oggetti a definire un analogo standard per questi sistemi. Questo standard è chiamato ODMG, come il gruppo (*Object Database Management Group*) che lo ha specificato ([Cat94b]).

Lo standard specifica un linguaggio ODL (*Object Definition Language*), per definire tipi oggetto e classi, ed un linguaggio OQL (*Object Query Language*), per definire interrogazioni. Specifica poi come chiamare questi linguaggi da programmi C++ e da Smalltalk per scrivere applicazioni complete.

Lo standard prescrive di dare un'implementazione singola di un tipo oggetto, in cui l'interfaccia è specificata in ODL e l'implementazione in un qualunque linguaggio; è prevista un'estensione verso l'approccio ad implementazioni multiple. Per ogni tipo oggetto è possibile definire una classe, che viene gestita con la tecnica dell'inserzione automatica. L'interfaccia di un tipo ne specifica i campi visibili e l'interfaccia dei metodi. Per gli attributi associativi viene specificato anche l'attributo che rappresenta l'associazione inversa. Le associazioni si modificano inserendo e cancellando coppie di valori associati, nello stile del Fibonacci. Tuttavia, la gestione del vincolo referenziale è analoga a quella del Galileo 97. Il linguaggio offre inoltre: sottotipi, definizione di tipi per ereditarietà, sottoclassi, quattro tipi di collezioni (insiemi, sequenze, multiinsiemi ed array), un meccanismo per la gestione delle eccezioni e delle transazioni. In Figura 1.5 viene riportato l'esempio delle persone e delle abitazioni scritto con la sintassi ODL.

Il linguaggio OQL è molto simile, anche nella sintassi, al linguaggio formato dagli operatori su sequenze definiti nella Sezione 1.4.2, e sarà ulteriormente illustrato in seguito.

1.18 Conclusioni

È stato presentato il Galileo 97 come esempio di linguaggio di programmazione per basi di dati con i meccanismi d'astrazione tipici dei modelli dei dati ad oggetti ed è stata fatta una breve analisi dei linguaggi O_2, UniSQL, ODL/OQL. La presentazione del Galileo 97 ha consentito di mostrare anche cosa vuol dire in generale trattare una base di dati con un linguaggio di programmazione e ciò tornerà utile nei prossimi capitoli, quando si vedranno i linguaggi dei sistemi commerciali per la gestione di basi di dati. Essendo il Galileo 97 un linguaggio che unifica i concetti di DDL e di DML, si è visto come descrivere in un ambiente le classi, i vincoli e le operazioni interessanti. I dati possono essere poi caricati interattivamente o definendo opportune funzioni. Una volta caricata la base di dati, essa può essere interrogata, in modo interattivo, usando un sottoinsieme del linguaggio. Questo è ciò che ci si aspetta di poter fare con un sistema per la gestione di basi di dati e le modalità viste con il Galileo 97 sono una esemplificazione di come ciò potrebbe avvenire. Nei sistemi commerciali

```
interface Abitazione
( extent Abitazioni )
{ readonly attribute String Categoria
  readonly attribute struct TIndirizzo { String Via; String Citta; } Indirizzo;
  relationship Set<PersonaVivente> Abitanti
            inverse PersonaVivente::Abitazione;
}

interface Persona
( extent Persone )
{ readonly attribute String Nome;
  readonly attribute Short AnnoNascita;
  relationship Persona Padre;
  relationship Persona Madre;
  Short Eta();
  Set<Persona> Antenati();
}

interface PersonaVivente: Persona
( extent PersoneViventi )
{ relationship Abitazione Dimora
            inverse Abitazione::Abitanti; }

interface PersonaDeceduta: Persona
( extent PersoneDecedute )
{ readonly attribute Short AnnoDecesso; }
```

Figura 1.5. Un esempio in ODL.

le cose saranno un po' diverse, ma il quadro di riferimento stabilito in questo capitolo sarà utile per confrontare meglio le proposte.

Esercizi

1. Discutere le differenze fra i tipi record e oggetto, e fra i valori record ed oggetto nel Galileo 97.
2. Discutere le differenze tra i valori sequenza e classe nel Galileo 97.
3. Discutere le nozioni di sottotipo e di tipo definito per ereditarietà nel Galileo 97.
4. Discutere l'utilità dei tipi **var** T nel Galileo 97.
5. Si considerino le seguenti definizioni Galileo 97:

```
let rec type
Persona <−>
    [ Nome       :string ;
      Indirizzo  :string ;
      Presentati := meth ():string is
                        self.Nome & " " & self.Indirizzo ];
let rec type
Impiegato <−> is Persona and
    [ Codice     :string ;
      Ditta      :string ;
      Presentati := meth ():string is
```

```
                    super.Presentati & " " & self.Ditta ];
let rec type
Studente <--> is Persona and
   [ Matricola   :string ;
     Presentati  := meth ():string is
                    super.Presentati & " " & self.Matricola ];

let Caio := mkPersona([Nome := "Caio"; Indirizzo :="Via Roma 2"]);
let Sempronio := mkStudente([ Nome := "Sempronio";
                              Indirizzo := "Via Roma 2";
                              Matricola := "267345"] );
```

(a) Dire quale metodo viene eseguito nella valutazione dell'espressione Caio.Presentati.

(b) Dire quale metodo viene eseguito nella valutazione dell'espressione Sempronio.Presentati.

(c) La notazione (Exp :T) denota il valore di Exp con il tipo T. Si supponga di porre:

```
let SempronioPersona := (Sempronio :Persona);
```

Dire quale metodo viene eseguito nella valutazione dell'espressione SempronioPersona.Presentati.

6. Si vogliono trattare informazioni su attori e registi di film. Di un attore o regista interessano il nome, che lo identifica, l'anno di nascita e la nazionalità. Un attore può essere anche un regista. Di un film interessano il titolo, l'anno di produzione, gli attori, il regista e il produttore. Due film prodotti lo stesso anno hanno titolo diverso. Si richiede di:

   (a) dare uno schema grafico;
   (b) dare uno schema Galileo 97;
   (c) scrivere in Galileo 97 la funzione per rendere un regista attore:
   mkAttoreDaRegista: Regista –> Attore
   (d) Scrivere un'espressione select in Galileo 97 per trovare:
      i. i nomi degli attori che hanno fatto almeno un film con il regista di nome Caio;
      ii. i nomi dei registi e il titolo del film, per ogni film in cui recita l'attore di nome Sempronio;
      iii. i nomi degli attori e dei registi del film con titolo Via col vento.

7. Si consideri lo schema grafico ottenuto dall'esercizio 3.1:

   (a) dare uno schema Galileo 97;
   (b) definire una funzione Galileo 97 che memorizzi il pagamento di una rata, con parametri CodiceMutuo, NumeroRata, Ammontare, DataVersamento.

8. Si ripeta l'esercizio precedente rappresentando l'inversa delle associazioni come proprietà memorizzata e non come metodo.

9. Si consideri il seguente schema Galileo 97, che descrive i dati sui clienti, auto ed incidenti, di interesse di una compagnia di assicurazioni.

```
let rec
Clienti class
    Cliente <->
        [ Codice            :string ;
          Nome              :string ;
          TotaleSpesa       :int ; % totale delle spese dei suoi incidenti %
          AutoPossedute     :seq string % targhe delle auto possedute %
        ] key (Codice)
and
Auto class
    Auto <->
        [ Targa             : string ;
          Modello           : string ;
          Proprietario      :CodiceCliente
        ] key (Targa)
and
Incidenti class
    Incidente <->
        [ Spesa             :int ;
          PercentualeColpa  :int ;
          Auto              :string % targa dell'auto %
    ];
```

   Si realizzi lo stesso schema in Galileo 97 rappresentando la diretta e l'inversa delle associazioni per aggregazione e la proprietà TotaleSpesa con un metodo.

10. Si ricorda che date due sottoclassi C_1 e C_2 di una classe C, diciamo che:

    (a) C_1 e C_2 soddisfano il vincolo di copertura se la loro unione è uguale a C;
    (b) C_1 e C_2 soddisfano il vincolo di disgiunzione se non hanno elementi in comune.

    In generale si possono quindi avere quattro tipi di situazioni:

    (a) copertura disgiunta o partizione (vincolo di copertura e di disgiunzione);
    (b) copertura non disgiunta (solo vincolo di copertura);
    (c) sottoinsiemi disgiunti (solo vincolo di disgiunzione);
    (d) sottoinsiemi non disgiunti (nessun vincolo).

    Si vuole sapere:

    (a) nel caso siano disponibili solo degli operatori di inserzione e cancellazione, quali mkT e dropT del Galileo 97, quale dei quattro casi precedenti si può modellare?
    (b) nel caso si abbia anche un operatore di estensione per muovere un oggetto dalla superclasse ad una sottoclasse, senza alterarne l'identità, quale inT del Galileo 97, quale dei quattro casi precedenti si può modellare?
    (c) nel caso si abbiano tutti gli operatori del caso precedente e anche la

possibilità di annullarne selettivamente alcuni, ad esempio di eliminare l'operatore mkC ponendo **let** mkC := nil, quali dei quattro casi precedenti si possono modellare?

Note bibliografiche

I linguaggi ad oggetti sono un'evoluzione dei linguaggi di programmazione per basi di dati [KA90, AB87]. Galileo, Galileo 97 e Fibonacci sono descritti in [Alb83], [ACO85], [Alb97], [AGO95] e [AAG00]. O_2 è descritto in [LR89, BDK92] ed ORION è descritto in [Kim90]. In [Kim90] sono presentati i sistemi per basi di dati ad oggetti prendendo come riferimento l'esperienza maturata dall'autore con la definizione e la realizzazione del linguaggio ORION e del relativo sistema di supporto. Per altri esempi di linguaggi per basi di dati ad oggetti si vedano [BM92, BDK92, Cat94a, Hug91, Kim90, KM94, Loo95].

BIBLIOGRAFIA

[AAG00] A. Albano, G. Antognoni, and G. Ghelli. View operations on objects with roles for a statically typed database language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567, 2000.

[AB87] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, 1987.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 39–51, Dublin, Ireland, 1993.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–439, 1995.

[Alb83] A. Albano. Type hierarchies and semantic data models. In *ACM SIGPLAN '83: Symposium on Programming Languages Issues in Software Systems*, pages 178–186, San Francisco, 1983.

[Alb97] A. Albano. Il manuale del Galileo 97. Servizio Editoriale Universitario, Pisa, 1997.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-oriented Database System. The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BM92] E. Bertino and L.D. Martino. *Sistemi di basi di dati orientate agli oggetti*. Addison-Wesley, Milano, 1992.

[Cat94a] R.G.G. Cattel. *Object Data Management. Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, Massachusetts, 1994.

[Cat94b] R.G.G. Cattel. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[Ghe91] G. Ghelli. A static type system for late binding overloading. In *Proc. of the Sixth Intl. ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1991.

[Hug91] J.G. Hughes. *Object-Oriented Databases*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[KA90] S. Khoshafian and R. Abnous. *Object Orientation, Concepts, Languages, Databases, User Interfaces*. J. Wiley & Sons, New York, 1990.

[Kim90] W. Kim. *Introduction to Object Oriented Databases*. MIT Press, Cambridge, Massachusetts, 1990.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall International, Inc., London, 1994.

[Loo95] M.E.S. Loomis. *Object Databases. The essentials*. Addison-Wesley, Reading, Massachusetts, 1995.

[LR89] C. Lecluse and P. Richard. The O_2 database programming language. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 411–422, Amsterdam, 1989.