Giorno 4
OOP

Sommario

Nel paradigma orientato agli oggetti, il sistema software è caratterizzato da un insieme di oggetti cooperanti, ognuno con i propri attributi (stato) ed i loro metodi (funzioni che agiscono (o meno, vd `static`) sullo stato). Vi sono due approcci diversi all'OOP, uno che si basa sulla manipolazione degli oggetti (object-based, usa subtyping strutturale e prototipi per l'ereditarietà), ed uno in cui gli oggetti sono istanze delle classi, (class-based, subtyping nominale, extends). Si studiano, inoltre, gli aspetti OOP di OCaml.

1 Il paradigma orientato agli oggetti

Nel paradigma orientato agli oggetti il sistema software è caratterizzato da un insieme di **oggetti cooperanti**; ogni oggetto è caratterizzato da:

- **Stato**: Gli attributi/proprietà/variabili dell'oggetto
- **Funzionalità**: i **metodi**/funzioni dell'oggetto.
- **Nome**: individua l'oggetto
- **Ciclo di vita**: Un oggetto può essere creato, riferito, disattivato
- **Locazione** di memoria.

Lo **stato del programma** consiste, idealmente, dell'insieme degli stati degli oggetti (anche se nella pratica a questo si aggiungono le strutture dati per il run-time support)

1.1 Classi ed Oggetti

Vi sono due approcci principali all'OOP:

- **Object-based** Come js (anche se dal 2016 supporta anche le classi)
- **Class-based** Come Java.

1.1.1 Approccio Object Based

Nell'approccio object based gli oggetti sono trattati nel linguaggio in modo simile ai record, con le seguenti differenze:

- I campi possono essere associati a funzioni
- Esiste **this**/self, tramite il quale un metodo può accedere ai campi dell'oggetto stesso.
- Js permette di modificare la struttura dell'oggetto dinamicamente, e.g. aggiungendo campi.

1.1.2 Approccio Class Based

Una classe definisce il contenuto degli oggetti di un certo tipo, e poi gli oggetti sono creati come istanze di una certa classe (e.g. **new**)

1.1.3 Differenze tra i due approcci

- L'approccio object-based consente al programmatore di lavorare con gli oggetti in modo più flessibile, ma rende difficile predire con precisione quale sarà il tipo di un oggetto (**ostacola controlli statici**)
- L'approccio class-based richiede al programmatore una maggiore disciplina e consente di fare **controlli di tipo statici** sugli oggetti: il tipo di un oggetto sarà legato alla classe da cui è stato istanziato (**nominal typing**, vd 1.2.2)

1.2 Inheritance e subtyping

La scelta tra object-based e class-based ha impatto sui meccanismo di ereditarietà e sottotipatura del linguaggio:

- I linguaggi **object-based** utilizzano la *prototype-based inheritance* e lo *structural (sub)typing.*
- I linguaggi **class-based** utilizzano la *class-based inheritance* ed il *nominal subtyping*

E voi direte: che sono? E io dirò:

1.2.1 Ereditarietà

L'ereditarietà (inheritance) è una funzionalità che permette di definire una classe/un oggetto a partire da un'altra/o esistente.

Ereditarietà per i linguaggi object-based: Per ogni oggetto si mantiene una lista di prototipi, ossia di oggetti da cui esso eredita funzionalità. Esiste un oggetto da cui tutti gli oggetti ereditano funzionalità, ed è solitamente detto **object**. In js `__proto__` è usato per risalire di un livello la catena dei prototipi.

Ereditarietà per i linguaggi class-based: I linguaggi class-based consentono di definire una classe come estensione di un'altra. La nuova classe eredita tutti i membri (valori e metodi) della precedente, con la possibilità di aggiungerne o ridefinirne alcuni.

1.2.2 Subtyping

- **Strutturale**: usato da linguaggi object based: un oggetto B è sottotipo di A se ha tutti i membri **pubblici** di A + eventualmente altri.
- **Nominale**: usato da linguaggi class-based: il tipo di un oggetto corrisponde alla classe da cui è stato istanziato: **nome della classe = tipo**. Un tipo-classe B è sottotipo di un tipo-classe A se la classe B è estensione di A.

Di nuovo, il meccanismo usato dal **class-based** è più rigoroso, mentre l'altro è più flessibile.

- Js ed OCaml adottano lo **structural** (anche se altri aspetti dell'ereditarietà di OCaml, non per gli oggetti, sono basati su simil-extends[1])

[1] qua immagino si faccia riferimento alla creazione di record con "with", ma potrei sbagliarmi.

1.3 OOP di OCaml

Un **oggetto** in ocaml è un valore fatto di campi e metodi. Gli oggetti possono essere creati direttamente (non ci sono classi). Il **tipo** di un oggetto è dato dai **metodi** che esso contiene (i **campi non influiscono** sul tipo). Esempio:

```
1  (* oggetto che realizza uno stack *)
2  let s = object
3     (* campo mutabile che contiene la rappresentazione dello stack*)
4     val mutable v = [0; 2] (* Assumiamo per ora inizializzato non vuoto *)
5     (* metodo pop *)
6     method pop =
7         match v with
8         | hd :: tl ->
9         v <- tl;
10        Some hd
11        | [] -> None
12    (* metodo push *)
13    method push hd =
14        v <- hd :: v
15 end ;;
```

Questo oggetto ha tipo:

```
1  val s : < pop : int option; push : int -> unit > = <obj>
```

Note sintattiche

- Nei metodi senza parametri non è necessario aggiungere ()
- Non c'è this, i campi sono visibili nei metodi.
- Per invocare i metodi si usa la # notation invece della dot notation.

1.3.1 Digressione: type weakening

Che sarebbe successo se avessi inizializzato il campo `v` dell'oggetto `s` con la lista vuota? (listing precedente, riga 4). Quale tipo verrebbe inferito per `s`?

Non sappiamo quale sia il tipo della lista, ma sappiamo che non potrà cambiare, una volta assegnato, quindi non è un tipo veramente polimorfo!

Il tipo degli elementi della lista è temporaneamente settato a `'_weak`, per poi **essere ricalcolato al primo utilizzo**.

```
1  val s : < pop : '_weak option; push : '_weak -> unit > = <obj>
```

1.3.2 Costruzione di oggetti tramite funzioni

Possiamo restituire un oggetto come risulatato di una funzione; tra i parametri potranno ad esempio esserci dei valori di inizializzazione:

```
1  (* funzione che costruisce oggetti inizializzati con init *)
2  let stack init = object
3     val mutable v = init
4
5     method pop =
6         match v with
7          | hd :: tl ->
8             v <- tl;
9             Some hd
10         | [] -> None
11    method push hd =
12        v <- hd :: v
13 end ;;
```

1.3.3 Polimorfismo di oggetti

Se definiamo una funzione il cui corpo accede ad un metodo di un **oggetto che non è ancora stato definito**, il tipo del metodo è inferito se possibile, se no sarà polimorfo.

Una **notazione** interessante è la seguente:

```
let area sq = sq#width * sq#width ;;
```

Il cui tipo è:

```
val area : < width : int; .. > -> int = <fun>
```

Le **parentesi angolate** indicano che l'oggetto su cui è chiamata la funzione deve avere **almeno** ($\Longrightarrow$ **subtyping strutturale**) un campo `width` di tipo intero.

Possiamo anche definire tipi utilizzando le parentesi angolate:

```
type shape = < area : float >
type square = < area : float; width : int >
```

È evidente che `square` è sottotipo di `shape`, e quindi ovunque è possibile usare un tipo `shape` possiamo usare al suo posto un tipo `square` (**principio di sostituzione**)

Operatore di coercion La conversione da `square` a `shape` non è però implicita: Si deve usare esplicitamente l'**operatore di coercion :>**

```
let lis2 = ( square 5 :> shape ) :: lis1 ;;
```

Si noti quindi che in OCaml i due concetti di **polimorfismo di oggetti** (funzioni che prendono oggetti di "almeno" un certo tipo) e **principio di sostituzione** (appena visto) sono trattati in modo diverso.

1.3.4 Perché OCaml non fa coercion implicita come Java

OCaml non fa **mai conversioni di tipo implicite**, mentre Java fa controlli dinamici di tipo svolti a runtime dall'interprete della JVM (resi più facili dal nominal subtyping).

1.3.5 Classi in OCaml

Per realizzare i meccanismi di ereditarietà, dato che usa lo stile *object-based*, OCaml dovrebbe usare i prototipi, il cui funzionamento è troppo complicato: per questo sono introdotti anche dei costrutti di **classe**: Una classe si definisce con **class** e si instanzia con **new**

```
class istack = object                          (* classe per stack di interi *)
   val mutable v = [0; 2]                      (* inizializzato non vuoto *)
   method pop =
       match v with
       | hd :: tl ->
           v <- tl;
           Some hd
       | [] -> None
   method push hd =
       v <- hd :: v
end ;;

let s = new istack ;;
```

Le classi possono essere **parametriche**:

```
class ['a] stack init = object                 (* classe polimorfa per stack *)
   val mutable v : 'a list = init              (* init e' parametro costruttore *)
   method pop = ....
   method push hd = ....
   end;;
```

Il tipo di un'istanza della classe è:

```
let s = new stack ["pippo"] ;;
(* => val s : string stack = <obj> *)
```

Nota: `string stack` è un **alias** per `<pop : string option; push : string -> unit>`
[Ereditarietà con inherit]