Università di Pisa, CdL in Informatica, A.A. 2014-15

Relazione del Progetto di Programmazione II

Giorgio Vinciguerra

5 giugno 2015

1. Interprete in OCaml

Come primo esercizio del progetto è stato chiesto di realizzare un interprete in OCaml per un linguaggio funzionale semplice che fornisca un costrutto di selezione *Try-with-in*. Il codice sorgente dell'Interprete è stato diviso in tre file:

1. `SyntacticDomains.ml` che contiene la definizione della sintassi astratta del linguaggio;
2. `SemanticDomains.ml` che contiene i tipi corrispondenti ai valori esprimibili del linguaggio, la definizione di ambiente e le funzioni che operano su di esso;
3. `SemanticFunctions.ml` con le funzioni di valutazione semantica.

Per rappresentare il costrutto Try-with-in, la sintassi è stata estesa nel seguente modo:

```
type exp = ... | Try of ide * exp * pat
and pat = CompClause of exp * exp * pat | BaseClause of exp * exp | Wildcard of exp
```

Pertanto un pattern è definito come una sequenza di clausole *composte*, terminata da una clausola *base* oppure da un *wildcard*. Il wildcard definisce il risultato di Try-with-in quando nessuna clausola è risultata vera (un comportamento simile al carattere underscore nel pattern matching di OCaml).

La valutazione di una espressione `Try(identificatore, expr, pattern)` crea un legame tra l'identificatore e il risultato dell'espressione che rimane attivo per tutta la valutazione del pattern. Se la prima espressione di una clausola del pattern (di base o composta) è vera, allora il risultato dell'intera espressione Try è dato dalla valutazione della seconda espressione. Altrimenti, se la clausola è composta, si continua con l'esecuzione del pattern successivo. La ricorsione termina quando si incontra un wildcard oppure quando l'ultima condizione del pattern è falsa, nel qual caso l'intera espressione Try è indefinita e viene sollevata una eccezione "Match failed". Il risultato è indefinito anche nel caso in cui la prima espressione di una clausola non è un valore booleano.

1.1. Implementare la regola di scoping dinamico

Se si volesse adottare la regola di scoping dinamico anziché statico, l'informazione che riguarda l'ambiente di *dichiarazione* di una funzione non serve più: l'ambiente non locale sarà quello presente al momento della *chiamata* a funzione. Dunque dobbiamo ridefinire `Funval` non più come una chiusura `efun = exp * eval env`, ma come una `efun = exp`.

Una volta fatto questo cambiamento dei domini semantici, la funzione `sem` assegnerà alle espressioni `Fun(i,bd)` il significato di `Funval(Fun(i,bd))`.
Invece, nel caso di espressioni `App(left,right)`, che indicano in sintassi concreta la chiamata a funzione `left(right)`, l'interpretazione produrrà la valutazione di `left` all'interno dell'ambiente corrente `r` (che dovrà restituire il corpo della funzione) e, successivamente, la valutazione del corpo ottenuto nell'ambiente nel quale è presente il legame tra il parametro formale `i` e il parametro attuale `right`. Possiamo tradurre quanto appena detto in:

```
let rec sem ((e:exp), (r:eval env)) = match e with ...
  | Fun(i,bd) -> Funval(e)
  | App(left,right) -> let f = sem(left,r) in
    begin match f with
      | Funval(Fun(formpar,body)) -> sem(body, bind(r,formpar,sem(right,r))
      | _ -> failwith "This is not a function, it cannot be applied"
    end
```

1.2. Istruzioni per eseguire il codice

Nella cartella `Interprete-OCaml` sono presenti due script. Il primo è `Run-interactive`, che lancia il toplevel di OCaml e carica i tre file sorgente, così da permettere un uso interattivo delle funzioni definite dall'Interprete. Il secondo script, `Run-tests`, lancia la test suite implementata nel file `Tests.ml` e stampa i risultati a schermo.

2. Modulo in Java

In questa seconda parte del progetto si chiedeva di progettare e realizzare un componente software per un sistema di *microblogging*. Il componente deve permettere, tramite una password, la gestione di un insieme di utenti registrati, soltanto ai quali è permessa la pubblicazione di messaggi di testo. Gli utenti sono identificati tramite un nickname. I messaggi sono caratterizzati da un *autore*, che deve necessariamente appartenere all'insieme degli utenti, un *testo* di lunghezza non superiore ai 140 caratteri e un *tag* che permette di etichettare il messaggio.

Al fine di consentire l'etichettatura un messaggio con zero, una o più di una stringa, come avviene nei principali servizi di microblogging, il tipo di dato Tag è stato definito ricorsivamente come un valore *nullo* o come una coppia (*stringa*, *oggetto di tipo Tag*).

La specifica forniva un interfaccia Java `SimpleTw`, che è stata usata come guida nell'implementazione della classe principale `myTw`. Per rendere più modulare il codice sono state introdotte anche le classi:

- `User`, le cui istanze, immutabili, hanno come unico campo stringa `nick`;
- `Tag`, le cui istanze, immutabili, hanno i campi `name`, di tipo stringa e `next`, di tipo `Tag`;
- `Tweet`, per rappresentare un singolo messaggio nel sistema. Mantiene riferimenti all'autore, ai tag e al testo del messaggio.

Tra i vari contenitori di oggetti che offre la *Java Collections Framework*, per terner traccia degli utenti registrati e dei messaggi è stata scelta la classe `ArrayList`.[1] In `myTw` sono dunque presenti le variabili d'istanza private `ArrayList<User> users` e `ArrayList<Tweet> tweets`.

I metodi che eliminano un messaggio o un utente dal sistema non rimuovono l'oggetto dalla collezione, ma lo sostituiscono con le (singole) istanze non modificabili delle classi `DeletedUser` e `DeletedTweet`. Questa scelta è stata fatta per due ragioni:

1. eliminare un oggetto da un `ArrayList` provoca il left-shift di tutti gli elementi che lo seguono.[2] In una collezione con migliaia di messaggi, come ci si aspetta da un servizio di microblogging, l'operazione provocherebbe un notevole rallentamento nell'esecuzione del programma;
2. ad ogni messaggio pubblicato viene associato un codice univoco che corrisponde alla posizione all'interno della collezione. Eliminando un oggetto in posizione i e facendo lo shift, tutti gli oggetti in posizione $j \geq i$ perderebbero la corrispondenza codice univoco-posizione.

Per il testing della classe principale è stato utilizzato il framework *JUnit*. Dopo ogni test case, per verificare che i metodi invocati tramite l'interfaccia non falsifichino l'invariante di rappresentazione, viene chiamato il metodo `repOk` su tutte le istanze di `myTw`. Con gli strumenti di analisi forniti dall'IDE utilizzato è stata misurata una statement coverage del 100%: possiamo dunque considerare la test suite *adeguata* a rivelare guasti in eventuali modifiche future alla classe principale.

2.1. Istruzioni per eseguire il codice

Insieme al codice sorgente del modulo è stato fornito un *Makefile*. Per compilare tutte le classi del modulo, posizionarsi tramite riga di comando nella cartella `Modulo-Java` e lanciare il comando `make`: i file con il bytecode verranno posizionati all'interno della cartella `classes`. Per compilare ed eseguire la test suite il comando da lanciare è `make test`.

[1] Altre classi, come `Vector`, offrono meccanismi di sincronizzazione che sono superflui per le finalità del progetto. Altre ancora, ad esempio `LinkedList`, non offrono un accesso per posizione in tempo costante, caratteristica necessaria per l'efficienza dei metodi che devono operare su indici come `delete(int code)` di `myTw`. `https://docs.oracle.com/javase/tutorial/collections/implementations/list.html`

[2] `https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#remove(int)`