LINGUAGGI

OCAML: INTERPRETE

Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

APPELLO D'ESAME 2017

Documentazione esame di linguaggi

Benedetta Amore
Andrea Erbisti
Giovanni Bellorio

SOMMARIO

OCAML

ANALISI PROGETTISTICA

Il progetto proposto dalla traccia può essere riassunto come segue:

Estendere l'interprete OCAML studiato durante le lezioni del corso di Linguaggi scegliendo una delle tre semantiche studiate: operazionale, denotazionale o iterativa.
La soluzione dovrà contenere il codice base imperativo e funzionale di blocchi, procedure e funzioni (escludendo l'implementazione degli oggetti).

L'estensione del linguaggio didattico dovrà comprendere:

- Il tipo stringa di caratteri: s
- La costante stringa nella semantica come sequenza di caratteri alfanumerici
- Il calcolo della lunghezza di una stringa: len (s)
- La concatenazione di stringhe: s1@s2
- L'operazione di sottostringa: substring (s, idx1, idx2)

La possibilità di introdurre ulteriori estensioni/operazioni prendendo ispirazione da operazioni note in linguaggi di scripting è lasciata allo studente.

L'estensione prevede anche l'implementazione del comando reflect (s) il quale prende in input una stringa s e chiama lo stesso interprete sulla stringa vista come sequenza di comandi.

OCAML

RAPPRESENTAZIONE

Il progetto è stato così raggruppato:

Per scrivere l'interprete abbiamo scelto la semantica denotazionale.
Per comodità abbiamo realizzato uno script che carica i vari file contenenti il codice dell'interprete ml. Per eseguire immediatamente l'interprete:

```
[MacBook-Pro-di-Giovanni:progetto Giovanni$ ocaml
        OCaml version 4.03.0

# #use "script.ml";;
```

Il progetto è suddiviso in sintassi, interfaccia ambiente, interfaccia memoria, interfaccia stack, domini semantici eval, in operazioni e in semantica. Descriviamo un po' meglio i file del progetto:

```
#use "1_Sintassi.ml";;

#use "2_Env_Interfaccia.ml";;
#use "2_Env_Semantica.ml";;
open Funenv;;

#use "3_Store_Interfaccia.ml";;
#use "3_Store_Semantica.ml";;
open Funstore;;

#use "4_Stack_Interfaccia.ml";;
#use "4_Stack_Semantica.ml";;
open SemPila;;

#use "4_StackM_Interfaccia.ml";;
#use "4_StackM_Semantica.ml";;
open SemMPila;;

#use "5_Eval.ml";;
#use "6_Operazioni.ml";;
#use "7_Semantica.ml";;
```

- **sintassi**, che contiene i domini sintattici e definisce la sintassi della semantica denotazionale da noi scelta.
- **env_interfaccia**, che contiene la signature dell'ambiente.
- **env_semantica**, che contiene l'implementazione dell'interfaccia dell'ambiente.
- **store_interfaccia**, che contiene la signature della memoria.
- **store_semantica**, che contiene l'implementazione dell'interfaccia dello store.
- **stack_interfaccia**, che contiene la signature dello stack NON modificabile.
- **stack_semantica**, che contiene l'implementazione dell'interfaccia dello stack non modificabile.
- **stackM_interfaccia**, che contiene la signature dello stack modificabile.
- **stackM_semantica**, che contiene l'implementazione dell'interfaccia dello stack modificabile.
- **eval**, che contiene i domini semantici (esprimibili eval, dichiarabili dval, memorizzabili mval) e le conversioni di tipo.
- **operazioni**, che contiene il typecheck, usato per testare di volta in volta il tipo dei parametri delle operazioni, e l'implementazione di tutte le operazioni.
- **semantica**, che contiene la definizione della semantica di ogni parte sintattica necessaria all'interprete.

OCAML

OPERAZIONI

Il progetto è stato così risolto:

Dopo una prima analisi progettista studiando tutti i requisiti abbiamo deciso di sviluppare un codice che appartenesse a un linguaggio imperativo. Tale ingloba quello funzionale richiedendo un'ambiente, una memoria e una espressione da valutare. Un'altra scelta tecnica e progettista è stata sul tipo di interprete: denotazionale.

Per soddisfare la richiesta di inserimento delle stringhe è stato aggiunto il tipo stringa al linguaggio, nella sintassi (type exp) e anche nella semantica (type eval esprimibile, type dval denotabile e type mval memorizzabile).

```
type exp =

  (* costanti *)
  | Eint of int
  | Ebool of bool
  | Estring of string
```

```
type eval =
    | Int of int
    | Bool of bool
    | Funval of efun
    | String of string
    | Novalue
```

Le operazioni sulle espressioni sono state implementate nel file Operazioni.ml il quale contiene anche il check di tipo e dichiarate nella sintassi con il loro numero di parametri. Nel linguaggio ci sono principalmente 3 tipi di dato: Int, Bool e ora anche String.
Per eseguire delle operazioni dobbiamo essere sicuri prima di tutto che gli operandi siano del tipo giusto altrimenti è impossibile eseguire qualcosa. Esempio: una somma di stringhe non può esistere e non può essere computata.

```
let typecheck (x, y) = match x with
    | "int" -> (match y with
      | Int(u) -> true
      | _ -> false)
    | "bool" -> (match y with
      | Bool(u) -> true
      | _ -> false)
    | "char" -> (match y with
      | String(u) -> true
      | _ -> false)
    | _ -> failwith ("not a valid type")
```

L'operazione Len restituisce un intero con la lunghezza della stringa passata come espressione. Per implementare tale operazione abbiamo utilizzato le librerie delle stringhe contenute in Ocaml (#use string.ml). In primo luogo abbiamo inserito il check di tipo e convertito la funzione len della libreria nel tipo String voluto dal nostro linguaggio.

```
let len x =
  if typecheck("char",x) then
    (match x with String(x) -> Int(String.length x))
  else
    failwith ("type error")
```

Stessa implementazione anche per l'operazione concat. Date due stringhe eseguire la concatenazione e restituire un'unica stringa. In libreria l'operazione concat è particolare perché è estesa a concatenare una lista di stringhe. Il trucco utilizzato è la trasformazione della stringa da concatenare in lista, unendola a una lista di stringhe vuote. Quindi verrà eseguita la concatenazione tra una stringa e una lista che effettivamente conterrà solo una stringa valida non nulla.

```
let concat (x, y) =
  if typecheck("char",x) && typecheck("char",y) then
    (match (x,y) with (String(u), String(w)) -> String(String.concat (u) [""];w]))
  else
    failwith ("type error")
```

L'operazione substring richiede vari controlli in seguito al check ti tipo. Si controlla che la lunghezza della stringa sia maggiore di 0 chiamando la funzione compare che controlla se un operando intero (lunghezza di s) è maggiore dell'altro (0). Dopodiché i due indici inseriti devono essere entrambi maggiori di 0, il primo minore del secondo ed entrambi minori della lunghezza della stringa.

Le specifiche richiedono che venga estratta la stringa dalla posizione i alla posizione j, mentre la substring della libreria lavora in modo diverso estraendo la stringa dalla posizione i per j caratteri. Quindi passiamo a tale funzione la posizione i come indici di start e j lo calcoliamo come differenza tra i due indici più 1.
La substring restituisce la sotto stringa richiesta se tutte le condizioni sono soddisfatte.

Tutte queste operazioni sono state inserite nella semantica del linguaggio e tali possono essere richiamate dinamicamente passando il numero corretto di parametri già aventi un tipo eval (non più exp) della semantica.

```
val sem : exp -> dval Funenv.env -> mval Funstore.store -> eval = <fun>
```

```
let rec sem (e:exp) (r:dval env) (s:mval store) = match e with
                | Estring(c)      -> String(c)
                | Len(c)          -> len(sem c r s)
                | Concat(c1, c2)  -> concat(sem c1 r s, sem c2 r s)
                | Substr(c, a, b) -> substr(sem c r s, sem a r s, sem b r s)
```

OCAML

REFLECT

La soluzione del comando Reflect. Tale comando presa in ingresso una stringa la converte in vere e proprie espressioni richiamando su di essa l'interprete e quindi la semantica. Tale comando è stato implementato nella semantica dei comandi del nostro linguaggio. La semantica dei comandi restituisce una memoria (store) definita precedentemente. Per visualizzare il risultato dobbiamo allocare e dichiarare una variabile di output del parse nella memoria e passare alla semantica dei comandi l'ambiente e la memoria su cui stiamo lavorando. La variabile da inizializzare e sulla quale verrà memorizzato il risultato è chiamata di default "result". Quello che dobbiamo implementare in sostanza è un vero e proprio parse, da stringa a exp.

```
val parseric : eval * exp SemMPila.stack * eval SemMPila.stack -> exp = <fun>
```

L'idea implementata per la soluzione della Reflect può essere così riassunta:

Esempio 1:

```
Sum(Eint a,Eint b)
Eint a,Eint b)
Eint b
)
```

Esempio 2:

```
Sum(Sum(Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Sum(Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
,Diff(Eint a,Eint b))
)
```

Il nostro parse ha bisogno di tre parametri: l'espressione (la stringa), uno stack dove memorizzare i risultati parziali (di tipo exp) e uno stack contenente l'espressione rimanente da convertire (di tipo eval). La funzione parse sfrutta la ricorsione dinamica. La soluzione implementata funziona interpretando la stringa in modo sequenziale. Sulla stringa non c'è nessun controllo iniziale di formattazione e di contenuto. Il processo riconosce i vari comandi interni grazie alla substring che controlla i caratteri di tale stringa con le vere e proprie espressioni. Una volta riconosciute cerca i suoi operandi (ogni operazione ha un numero di operandi diverso) che possono essere ancora operazioni che restituiscono un risultato base. Per fronteggiare a tale operazione il controllo sul numero di virgole all'interno dell'operazione è limitato. Occorre utilizzare lo stack. Si effettua la push del parse della stringa rimanente fino ad arrivare a un passo base, cioè fino a quando non si trova un intero, stringa o booleano. La conversione avviene digitando l'operazione sotto forma di exp. Per ottenere il contenuto dello stack si utilizza la funzione topandpop che automaticamente esegue il top e il pop dello stack.

```
val topandpop : 'a SemMPila.stack -> 'a = <fun>
```

I passi base sono implementati in modo tale da concludere parzialmente l'operazione salvando sullo stack il risultato base estratto controllando la posizione della "," e salvando sullo stack ausiliario la stinga rimanente da scansionare.
Nel caso di interi e booleani si utilizzano anche le funzioni di libreria per effettuare la conversione da stringa a intero o da stringa a booleano.

```
val convert : eval -> int = <fun>
```

```
val convert_bool : eval -> bool = <fun>
```

Analizzando passo a passo il problema è necessario inserire alcune condizioni per far continuare l'operazione: nel caso in cui si trovi una parentesi tonda ")" di troppo dovuta alle varie operazioni nidificate, alle "," che dividono gli operandi di una operazione.

```
else if equals(substr(String(n),Int(0),Int(0)),String(",")) then
  parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,stackstr)

else if equals(substr(String(n),Int(0),Int(0)),String(")")) && (String.length(n)!=1) then
  parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,stackstr)
```

OCAML

LIMITAZIONI

La soluzione implementata implica l'utilizzo delle pile stack. Per inizializzare una pila stack modificabile è necessario specificare il tipo di pila e il numero di "celle" da riservare per "memorizzare i dati".

```
val emptystack : int * 't -> 't stack
```

Per quanto riguarda il risultato del comando reflect possiamo essere sicuri che è al più uno mentre non possiamo prevedere il numero di operazioni che possono essere valutate e quindi impostiamo di default un massimo di 100 operazioni. Potenzialmente la soluzione risolve un numero infinito di ricorsioni ma impostiamo un limite.

I warning, invece, sono dovuti al non utilizzo delle variabili dichiarate con let.

È possibile stampare il risultato della Reflect solo nella variabile "result" solo se inizializzata.

```
let com = Reflect(Estring s);;

let (l,sigma) = allocate(emptystore(Undefined), Undefined);;
let rho = bind (emptyenv(Unbound), "result", Dloc l);;
let sigma1 = semc com rho sigma;;

sem (Val(Den "result")) rho sigma1;;
```

OCAML
ESEMPI

```
################ESEMPI SEMANTICA#############################################################

let l = "CIAO";;
let l1 = "COME";;
let l2 = "VA";;

sem (Estring l) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l)) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l1)) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Concat(Estring l1, Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Concat(Estring l, Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 1, Eint 3)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 1, Eint 2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 0, Eint 1)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 4, Eint 10)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 10, Eint 12)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 3)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 0)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 0)) (emptyenv Unbound) (emptystore Undefined);;
```

```
################ESEMPI REFLECT#########################################

let s = "Sum(Eint 1,Sum(Eint 5,Eint 4))";;
let s = "Sum(Sum(Eint 10,Eint 1),Diff(Eint 2,Eint 1))";;
let s = "Sum(Len(Estring CIAO),Eint 1)";;
let s = "Len(Estring CIAO)";;
let s = "Concat(Estring CIAO,Estring CIAO)";;
let s = "Substr(Estring CIAO,Eint 0,Eint 1)";;
let s = "Substr(Estring CIAO,Sum(Eint 0,Eint 1),Sum(Eint 1,Eint 1))";;

let com = Reflect(Estring s);;

let (l,sigma) = allocate(emptystore(Undefined), Undefined);;
let rho = bind (emptyenv(Unbound), "result", Dloc l);;
let sigma1 = semc com rho sigma;;

sem (Val(Den "result")) rho sigma1;;
```