Giorno 2
Implementazione di Interpreti

Sommario

Si presentano un interprete di espressioni aritmetiche, con cenni sulla struttura ed implementazione di scanner e parser, un interprete del λ-calcolo ed uno di una versione didattica di Caml.

1 Introduzione

1.1 Ciclo di interpretazione

Le fasi dell'interpretazione sono:

- Scanner: Analisi lessicale, restituisce token list
- Parser: Analisi grammaticale, genera AST
- Type checker: Controllo dei tipi
- Interprete: restituisce il risultato.

Scanning e parsing possono essere realizzati a livello di intero programma o di singole istruzioni/espressioni, e lo stesso vale anche per le altre fasi.

1.1.1 Esempio: NodeJS vs OCaml

Node.js controlla prima la sintassi di tutto il programma, e poi interpreta:

`console.log(1); console.log(2;` → Non stampa 1 prima di vedere che manca la parentesi.

Mentre il toplevel di OCaml esegue il parsing sulle singole espressioni:

`let x = 10;; lett y = 20;;` → Esegue comunque la prima espressione.

1.2 Interprete di espressioni aritmetiche

1.2.1 Scanner

- Definiamo i token come tipo union:

  ```
  type token =
  Tkn_NUM of int | Tkn_OP of string | Tkn_LPAR | Tkn_RPAR | Tkn_END;;
  ```

- Creiamo un'eccezione `ParseError`:

  ```
  exception ParseError of string * string;;
  ```

- Che utilizzeremo nel seguente modo:

  ```
  raise (ParseError(``Tokenizer'', ``unknown symbol''^c))
  ```

  ...all'interno della funzione ricorsiva `tokenize`, che scansionando i caratteri costruisce una lista di token ad essi corrispondenti, e.g.

```
match c with
   |" " -> tokens | "(" -> Tkn_LPAR :: tokens | ")" -> Tkn_RPAR
   | "+" | "-" | "*" | "/" -> (Tkn_OP c) :: tokens
   | "0" | "1" | ... | "9" -> Tkn_NUM (cifre consecutive accorpate)
```

Problema: Implementazione non tail-recursive, ma vabbè.

1.2.2 Parser

Si realizza un *parser a discesa ricorsiva*:

- Si crea una funzione per ogni categoria sintattica
  (nell'esempio: $Exp ::= Term[\pm Exp]$, $Term ::= Factor[* Term \mid /Term]$, $Factor ::= n \mid (Exp)$)
- Le funzioni create sono mutuamente ricorsive (`and`)
- L'AST corrisponderà all'albero delle chiamate

Implementazione

- Si passa la lista dei token per riferimento
- Si definiscono due funzioni speciali: **lookahead**, che permette di vedere qual è il prossimo token da processare senza eliminarlo, e **consume**, che NON restituisce ma elimina il primo token della lista.

```
let tokens = ref (tokenize s) in

let lookahead () = match !tokens with
  | [] -> raise (ParseError("Parser", "lookahead error"))
  | t :: _ -> t

in consume () = match !tokens with
  | [] -> (ParseError("Parser", "consume error"))
  | t::tkns -> tokens := tkns
```

- Ad esempio, per gestire le operazioni binarie delle espressioni:

```
let rec exp () = (* si noti che la funzione ha tipo unit *)
  let t1 = term() in
      match lookahead () with
          | Tkn_OP "+" -> consume() ; Op (Add t1, exp())
          | Tkn_OP "-" -> consume() ; Op (Sub t1, exp())
```

  La chiamata ricorsiva ad `exp` in `Op (Add t1, exp())` fa un passo della costruzione dell'AST.

- L'AST completo viene ottenuto facendo: `let ast = exp ()` e in seguito controllando se una chiamata di `lookahead()` restituisce solo il token di fine. In caso contrario non tutti i token sono stati valutati, e c'è stato perciò un parse error.

1.2.3 Nota sui parser

L'implementazione dei parser non è sempre così semplice; si usano solitamente dei *parser generator* che li creano per noi (non argomento del corso). Nei linguaggi che implementeremo partiremo perciò dall'AST, e non implementeremo un parser.

1.2.4 Interprete

Vi sono due approcci alla definizione della semantica di un linguaggio:

- Approccio **big-step**: La relazione di transizione descrive in un solo passo *l'intera computazione.* (le singole operazioni sono descritte nell'albero di derivazione della transizione)

$$n \to n \qquad \frac{E_1 \to n_1 \quad E_2 \to n_2 \quad n_1 \text{ op } n_2 = n}{E_1 \text{ op } E_2 \to n}$$

- Approccio **small-step**: Ogni passo della relazione di transizione esegue *una singola operazione.*

$$n \to n \qquad \frac{E_1 \to E_1'}{E_1 \text{ op } E_2 \to E_1' \text{ op } E_2}$$

$$\frac{E_2 \to E_2'}{n \text{ op } E_2 \to n \text{ op } E_2'} \qquad \frac{n_1 \text{ op } n_1 = n}{n_1 \text{ op } n_2 \to n}$$

Metodo sistematico per passare da semantica a codice

- Si fa pattern matching sull'espressione, creando un caso per ogni tipo di nodo dell'AST (nel nostro caso i nodi possono essere *operazioni* o *valori*, le espressioni sono i sottoalberi dell'AST)
- Si verificano le pre-condizioni delle varie regole (non assiomi), chiamando ricorsivamente l'interprete.
- Quando le pre-condizioni sono verificate, si calcola il risultato della transizione.

[Esempi di interprete big-step vs small-step: listing `[16]` e `[18]` di "Introduzione allo sviluppo di interpreti"]

2 Interprete del λ-calcolo

2.1 Sintassi

$$e \coloneqq x \mid \lambda x.e \mid e\ e$$

```
1  type id = string;;
2  type exp = Var of id | Lam of id * exp | App of exp * exp
```

2.2 Semantica

Realizziamo una versione deterministica della semantica che abbiamo visto:

$$(\lambda x.e_1)e_2 \to e_1\{x \coloneqq e_2\}$$

$$\frac{e_1 \to e'}{e_1 e_2 \to e' e_2} \qquad \frac{e_2 \to e'}{e_1 e_2 \to e_1'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Al fine di realizzare un interprete del λ-calcolo dobbiamo anzitutto implementare l'operazione più complessa della semantica del λ-calcolo: la **capture-avoiding substitution**.
Partiamo dalla definizione formale:

$$\begin{aligned}
x\{x \coloneqq e\} &\equiv e \\
y\{x \coloneqq e\} &\equiv y, \text{ se } y \neq x \\
(e_1 e_2)\{x \coloneqq e\} &\equiv (e_1\{x \coloneqq e\})(e_2\{x \coloneqq e\}) \\
(\lambda y.e_1)\{x \coloneqq e\} &\equiv \begin{cases} \lambda y.(e_1\{x \coloneqq e\}) & \text{se } y \neq x \text{ e } y \notin FV(e) \\ \lambda z.((e_1\{x \coloneqq z\})\{x \coloneqq e\}) & \text{se } y \neq x \text{ e } y \in FV(e), \text{ z fresca} \end{cases}
\end{aligned}$$

E notiamo che possiamo implementare senza problemi questa definizione tramite il pattern matching, ma è necessario definire un metodo per creare variabili "fresche" e per costruire l'insieme (per i nostri scopi basta una lista) delle variabili libere di un'espressione.

2.2.1 Implementazione di FV

La definizione dell'insieme delle variabili libere è la seguente:

$$FV(x) = \{x\}$$
$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(\lambda x.e) = FV(e) \setminus \{x\}$$

In OCaml:

```
let rec fvs e =
    match e with
        | Var x -> [x]
        | Lam (x, e) -> List.filter (fun y -> x <> y) (fvs e)
        | App (e_1 e_2) -> (fvs e_1) @ (fvs e_2)
;;
```

2.2.2 Generatore di id freschi

```
let newvar =
    let x = ref 0 in
        fun () ->
            let c = !x in
            incr x;
            "v"^(string_of_int x)
```

Ogni volta che è chiamata genera una variabile vn, partendo da `v0`. Riesce a mantenere x poiché è un riferimento ed è sempre nella chiusura della funzione.

A questo punto abbiamo tutti i mezzi per implementare la capture avoiding substitution, realizzando tutti i casi tramite il pattern matching (si veda il listing `[6]` di "Un interprete del λ-calcolo")

La semantica è implementata usando il *metodo sistematico* descritto in 1.2.4 (si veda il listing `[8]`). È interessante notare che non è possibile implementare precisamente la semantica non deterministica che abbiamo definito; si deve dare priorità ad una delle regole di valutazione dell'applicazione, e in base all'ordine dei pattern si può scegliere se dare priorità alla riduzione o all'applicazione funzionale.

L'implementazione completa dell'interprete mostrato a lezione è reperibile qua:
`https://www.cs.umd.edu/class/spring2019/cmsc330/lectures/interp.ml`

3 MiniCaml

3.1 Ambiente

Perché il λ-calcolo non ha l'ambiente? Nel λ-calcolo l'ambiente non era necessario, poiché i binding tra identificatori e valori erano implementati tramite la sostituzione

A differenza dall'interprete del λ-calcolo, per realizzare l'interprete del *MiniCaml* serve un'implementazione dell'ambiente. Potremmo realizzarlo come:

- Lista di coppie $(nome, valore)$, (non ciò che useremo)
- **Funzione Polimorfa** $\Gamma : Ide- > Value \cup \{Unbound\}$;

  $\Gamma(x)$ denota il valore v associato ad x nell'ambiente, oppure, se non esiste il binding, il valore speciale *Unbound*. La funzione Γ può essere estesa con un legame, notazione $\Gamma[x = v]$, e:

$$\Gamma[x = v](y) = \begin{cases} v & \text{se } y = x \\ \Gamma(y) & \text{se } y \neq x \end{cases}$$

```
(* 't = tipo dei valori esprimibili *)
type 't env = ide -> 't;;
(* ambiente vuoto *)
let emptyenv = fun x -> Unbound;;

(* aggiornamento ambiente s con (x, v) *)
let bind s x v =
   fun i -> if (i = x) then v else (s i);;
```

Nota *Il tipo dell'ambiente è polimorfo perché sarà definito in mutua ricorsione con il tipo dei valori esprimibili (`Closure` e `RecClosure`, vedi 3.4.2)*

3.2 Il linguaggio

```
type ide = string
type exp =
    | CstInt of int                 (* costante Int *)
    | CstTrue                       (* costante True *)
    | CstFalse                      (* costante False *)
    | Sum of exp * exp
    | Diff of exp * exp
    | Prod of exp * exp
    | Div of exp * exp
    | Eq of exp * exp
    | Iszero of exp
    | Or of exp * exp
    | And of exp * exp
    | Not of exp
    | Den of ide                    (* Entita' denotabile (variabile) *)
    | Ifthenelse of exp * exp * exp
    | Let of ide * exp * exp        (* Dichiarazione di ide: modifica ambiente *)
    | Fun of ide list * exp         (* Astrazione di funzione *)
    | Apply of exp * exp list       (* Applicazione di funzione *)
```

Si noti che `Fun` ha come "ramo argomenti" una lista; questa è la lista dei parametri. In realtà questa parte sarà semplificata, e useremo funzioni di una sola variabile.

3.3 Espressioni

Per il momento lavoriamo solo con le espressioni, (fino a riga 16 del listing in 3.2). Dato che abbiamo più di un tipo di dato esprimibile, dobbiamo fare del **typechecking**.

Valori esprimibili / descrittori di tipo

```
(* Possibili risultati delle
   valutazioni di espressioni *)
type evT = Int of int
         | Bool of bool
         | Unbound
```

Tipi esistenti

```
type tname =
 | TInt
 | TBool
 | ...
 | ...
```

Tramite il typechecking vogliamo vedere se un descrittore di tipo ha effettivamente il tipo che vogliamo: controlliamo perciò una **coppia `(type, typeDescriptor)`**.

Codice del typechecking

```
let typecheck (type, typeDescriptor) =
    match type with
    | TInt ->
        (match typeDescriptor with
            | Int(u) -> true
            | _ -> false)
    | TBool ->
        (match typeDescriptor with
            | Bool(u) -> true
            | _ -> false)
    | _ -> failwith ("not a valid type")
    ;;
```

- Se il tipo è int:
  - Se il typeDescriptor matcha un intero, allora il tipo corrisponde
  - Se no non corrisponde
- Se il tipo è bool:
  - Se il typeDescriptor matcha un booleano, allora il tipo corrisponde
  - Se no non corrisponde

Implementazione delle operazioni di base Le operazioni sono implementate come funzioni:

```
let is_zero x = match (typecheck(TInt,x), x) with
    | (true, Int(y)) -> Bool(y=0)
    | (_, _) -> failwith("run-time error");;

let int_eq(x,y) =
    match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Bool(v = w)
    | (_,_,_,_) -> failwith("run-time error ");;

let int_plus(x, y) =
    match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
    | (true, true, Int(v), Int(w)) -> Int(v + w)
    | (_,_,_,_) -> failwith("run-time error ");;
```

Quindi la funzione `eval`, che prenderà come parametri un'espressione el'ambiente, sarà del tipo:

```
let rec eval (e:exp) (amb: evT env) =
    match e with
        | CstInt(n) -> Int(n)
        | CstTrue -> Bool(true)
        | CstFalse -> Bool(false)
        | Iszero(e1) -> is_zero(eval e1 amb)
        | Eq(e1, e2) -> int_eq((eval e1 amb), (eval e2 amb))
        | Sum(e1, e2) -> int_plus ((eval e1 amb), (eval e2 amb))
        | ...
```

Binding di identificatori
Per coprire anche il caso di `e = Den i` (entità denotabile), aggiungiamo anche

```
Den (i) -> amb i
```

Che restituisce il valore associato all'identificatore `i` nell'ambiente `amb`.

If then else L'if viene implementato, a differenza delle altre espressioni, con una strategia *non-eager* (per evitare di valutare il ramo che non ci interessa, che e.g. potrebbe non terminare)

```
 Ifthenelse(cond,e1,e2) ->
let g = eval cond amb in
match (typecheck("bool", g), g) with
| (true, Bool(true)) -> eval e1 amb
| (true, Bool(false)) -> eval e2 amb
| (_, _) -> failwith ("nonboolean guard")
```

3.4 Variabili, funzioni

3.4.1 Let: blocco

Il `let x = e1 in e2` valuta l'espressione `e1`, trasformandola nel valore `v`, e calcola l'espressione `e2` a partire dall'ambiente Γ a cui aggiungo il binding `(x, v)`

$$\frac{\Gamma \triangleright e_1 \to v_1 \qquad \Gamma[x = v_1] \triangleright e_2 \to v_2}{\Gamma \triangleright Let(x, e_1, e_2) \to v_2}$$

(dove $\Gamma \triangleright e$ significa "e, valutata nell'ambiente Γ")

Nota L'operazione di estensione **corrisponde** alla push di un record di attivazione. Per come è implementato l'ambiente, l'estensione crea una nuova *funzione ambiente*, costruita a partire dall'ambiente precedente, che sarà "dimenticata" all'uscita del blocco (pop);

```
let rec eval((e: exp), (amb: evT env)) =
   match e with
   ...
   ...
   | Let(i, e, ebody) ->
       eval ebody (bind amb i (eval e amb))
```

3.4.2 Funzioni

Consideriamo le funzioni con un solo parametro:

`Fun of ide * exp` `Apply of exp * exp`

Dato che adesso dobbiamo esprimere anche tipi funzione, dobbiamo **estendere i valori esprimibili**:

```
type evT = Int of int | Bool of bool | Unbound | Closure of ide * exp * evT env
```

Una *chiusura* avrà perciò la forma `(x, e, env)`, dove `x` è il parametro formale della funzione, `e` è il corpo della funzione e `env` è l'ambiente che era attivo alla definizione della funzione (Γ_{decl}).

Dichiarazione di funzione:

$$\Gamma \triangleright Fun(x, e) \to Closure(\texttt{"x"}, e, \Gamma)$$

Vado a **ripescare la funzione** dall'ambiente (chi è "Var"? forse Den boh possibile errore nelle slide):

$$\Gamma \triangleright Var(\texttt{"f"} \to Closure(\texttt{"x"}, body, \Gamma_{fDecl}))$$

Applicazione di funzione, dove v_a è il "valore attuale" ottenuto valutando il parametro attuale:

$$\frac{\Gamma \triangleright arg \to v_a \qquad \Gamma_{fDecl}[x = v_a] \triangleright body \to v}{\Gamma \triangleright Apply(Den(\texttt{"f"}), arg) \to v}$$

Spiegazione di Milazzo[1]:

> *Il corpo della funzione viene valutato nell'ambiente ottenuto legando il parametro formale al valore del parametro attuale nell'ambiente nel quale era stata valutata l'astrazione.*

[1] meglio di mille spiegazioni mie – (dalle slide "MiniCAML_parte1")

Implementazione: funzioni

```
let rec eval (e: exp) (amb: evT env) =
    match e with
        | ...
        | Fun(i, a) -> Closure(i, a, amb)
        | Apply(Den(f), eArg) ->
            let fclosure = amb f in
            (match fclosure with
                | Closure(arg, fbody, fDecEnv) ->
                    let aVal = eval eArg amb in
                    let aenv = bind fDecEnv arg aVal in
                        eval fbody aenv
                | _ -> failwith("non functional value"))
        | Apply(_,_) -> failwith("Application: not first order function") ;;
```

Nota sullo scoping dinamico La differenza tra **scoping statico** e **scoping dinamico** è che con lo scoping statico l'ambiente è costruito in base *alla struttura del programma* (come nel nostro caso), mentre con lo scoping dinamico l'ambiente è costruito in base *al flusso del codice*; quest'ultimo è più semplice da implementare, basterebbe infatti usare l'ambiente corrente invece di Γ_{fDecl}.

Devo perciò modificare `evT`, sostituendo `Closure` con `Funval of ide * exp`, non c'è più bisogno dell'ambiente.

3.5 Funzioni ricorsive

3.5.1 Let rec

Definiamo un costrutto `let rec`, che allo stesso tempo ha la funzione di creare un blocco e di creare una funzione con nome:

```
type exp =
    ...
    | Letrec of ide * ide * exp * exp
```

In particolare: `Letrec("f", "x", fbody, letbody)`, dove f è il nome della funzione, x è il parametro formale, *fbody* è il corpo della funzione, e *letbody* è il corpo del let.

3.5.2 Valori esprimibili

Dobbiamo estendere ulteriormente i valori esprimibili:

```
type evT = | Int of int | Bool of bool
           | Unbound | Closure of ide * exp * evT env
           | RecClosure of ide * ide * exp * evT env
```

$\Longrightarrow$ `RecClosure(funName, param, funBody, staticEnvironment)`

Si noti che RecClosure in realtà esprime semplicemente la chiusura di una funzione con nome, che può benissimo anche non essere ricorsiva.

3.5.3 Codice dell'interprete

Il codice dell'interprete è simile a quello delle funzioni anonime non ricorsive, ma valuto `letbody` nell'ambiente $\Gamma[f = RecClosure(f, i, fBody, amb)]$, per rendere possibile la ricorsione:

Dichiarazione di funzione ricorsiva:

```
...
| Letrec(f, i, fBody, letBody) ->
    let benv =
        bind amb f (RecClosure(f, i, fBody, amb))
            in eval letBody benv
...
```

Applicazione di funzione ricorsiva

```
| Apply(Den f, eArg) ->
    let fclosure = amb f in
        match fclosure with
            | Closure(arg, fbody, fDecEnv) ->
                ...
                ...
            | RecClosure(f, arg, fbody, fDecEnv) ->
                let aVal = eval eArg amb in
                    let rEnv = bind fDecEnv f fclosure in
                        let aEnv = bind rEnv arg aVal in
                            eval(fbody, aEnv)
        | _ -> failwith("non functional value")
| Apply(_,_) -> failwith("not function")
```

Unica differenza dalle funzioni non ricorsive: bind di f nell'ambiente `(9)` prima del bind dell'argomento.

3.6 Funzioni di ordine superiore

Per ora non possiamo passare funzioni come argomento, ossia trattarle come valori di prima classe.

Per ora, infatti, (si veda riga `(5)` del listing in 3.4.2) la apply deve avere la forma `Apply (Den (f), eArg)`, se no si ha un errore a runtime (*"not first order function"*).

Per risolvere basta:

- Accettare anche espressioni `eF`
- Valutarle prima di fare il patternmatching con le chiusure.

```
| Apply(eF, eArg) ->
    let fclosure = eval eF amb in
        (match fclosure with
            | Closure(arg, fbody, fDecEnv) ->
                let aVal = eval eArg amb in
                    let aenv = bind fDecEnv arg aVal in
                        eval fbody aenv
            | RecClosure(f, arg, fbody, fDecEnv) ->
                let aVal = eval eArg amb in
                    let rEnv = bind fDecEnv f fclosure in
                        let aenv = bind rEnv arg aVal in
                            eval fbody aenv
            | _ -> failwith("non functional value")) ;;
```

Esempio Run-Time Stack

Si fornisce adesso un'idea di come funzionano i record di attivazione per i linguaggi non interpretati:

- Un record di attivazione consiste di uno Static Link (catena statica), che punta al record di attivazione del chiamante, e di un valore (valore aggiunto all'ambiente)
- Se il valore non è funzionale, si mantiene nello stack
- Se il valore è funzionale (**non ricorsivo**), si mantiene una coppia `<f, A>`, dove f è un puntatore al codice della funzione, ed A è una copia dello static link, ossia un puntatore all'ambiente in cui dobbiamo eseguire la funzione.
- Se il valore funzionale è **ricorsivo**, non possiamo eseguire la funzione nel record di attivazione puntato dallo static link: dobbiamo copiare un puntatore al record di attivazione corrente, in cui è definito il nome della funzione.

Problema: Che succede se una funzione cerca di restituire un valore funzionale?
Il RdA che contiene l'ambiente a cui quella funzione fa riferimento potrebbe essere stato cancellato!
$\Longrightarrow$ **Soluzione**: mettiamo un flag ai record di attivazione, che indica se quel RdA è in uso; in tal caso non lo cancelliamo.