

Programmazione II

Luca De Paulis

1 gennaio 2021

INDICE

I LINGUAGGI DI PROGRAMMAZIONE

1	SINTASSI DI OCAML	4
1.1	Valori ed espressioni	4
1.2	Costrutto let	4
1.3	Funzioni	5
1.4	Liste	8
1.5	Nuovi tipi di dato	9
2	SINTASSI E SEMANTICA	11
2.1	Sintassi	11
2.2	Meccanismi di inferenza	11
2.3	Semantica statica	12
2.4	Semantica dinamica	13
2.5	Tipi di dato	16
2.6	Type-system	20
2.6.1	Implementazione in OCaml di un type system	21
2.7	Chiamate a funzione	22
2.7.1	Blocchi	23
2.7.2	Funzioni e procedure	24
2.8	Passaggio dei parametri	28
2.9	Linguaggio didattico	30
3	SEMANTICA DEI LINGUAGGI OBJECT-ORIENTED	37
3.1	Classi e metodi	37
3.2	Java Virtual Machine	39
3.3	Garbage collection	41

Parte I

LINGUAGGI DI PROGRAMMAZIONE

1

SINTASSI DI OCAML

1.1 VALORI ED ESPRESSIONI

OCaml è un linguaggio multiparadigma derivato dal linguaggio funzionale CaML e dai suoi antenati nella famiglia di ML. Essendo alla base un linguaggio funzionale, un programma OCaml è un'espressione che può essere valutata ad un *valore*, ovvero ad un'espressione che non deve essere valutata ulteriormente.

Useremo la notazione $\langle \text{exp} \rangle \Rightarrow v$ oppure la notazione $\text{Eval}(\text{exp}) = v$ per dire che l'espressione exp viene valutata al valore v .

Il primo metodo per creare espressioni è il costrutto **let**. Ad esempio l'espressione

```
let x = 42;;
```

ci permette di *legare* al nome della variabile x il valore **42**. Questo costrutto ci consente tuttavia di definire espressioni più complesse:

```
let x = 7*3
in x*x;;
```

Questa espressione lega alla variabile x il valore **21**, cioè il valore a cui l'espressione $7*3$ viene valutata; inoltre, questo legame è *locale* all'espressione che segue la parola chiave **in**. Il valore dell'espressione è quindi **441**.

1.2 COSTRUTTO **let**

Il costrutto **let** ci permette inoltre di introdurre una nozione di *scope*: nel seguente codice la variabile x è definita e visibile dentro il **let** più esterno, mentre la variabile y esiste ed è visibile solo all'interno del secondo **let**.

```
let x = 42
in x + (let y = "3110"
in int_of_string y);;
```

Il valore di questa espressione è **3152**, in quanto il valore dell'espressione **let** interna è il numero intero **3110**, mentre il valore dell'espressione più esterna è dato dalla valutazione di $x + 3110$, che restituisce **3152**.

Regola di valutazione del costrutto **let**

Studiamo ora la regola formale di valutazione del **let**:

$$\frac{\text{Eval}(e1) = v' \quad \text{subst}(e2, x, v') = e2' \quad \text{Eval}(e2') = v}{\text{Eval}(\text{let } x = e1 \text{ in } e2) = v}.$$

Questa regola ci dice che la valutazione dell'espressione **let** $x = e1$ **in** $e2$ è v se:

- la valutazione di $e1$ è un valore v' ;

- sostituendo il valore v' al posto di tutte le occorrenze libere di x nell'espressione $e2$ otteniamo l'espressione $e2'$;
- la valutazione di $e2'$ è esattamente il valore v .

Esempio 1.2.1. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 2 + (5 * 3) in x + 10;;
```

- (1) Valutiamo l'espressione $2 + (5 * 3)$, il cui valore è 17.
- (2) Sostituiamo 17 al posto di ogni occorrenza libera di x nell'espressione $x + 10$, ottenendo l'espressione $17 + 10$.
- (3) Valutiamo quest'ultima espressione, ottenendo 27, che è il valore dell'espressione iniziale.

Esempio 1.2.2. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 12 - 3
  in let y = 2 + x
      in x == y;;
```

- (1) Valutiamo l'espressione $12 - 3$, il cui valore è 9.
- (2) Sostituiamo 9 al posto di ogni occorrenza libera di x nell'espressione `let y = 2 + x in x == y`, ottenendo l'espressione `let y = 2 + 9 in 9 == y`.
- (3) Valutiamo quest'ultima espressione: siccome anch'essa è un costrutto `let` bisogna applicare nuovamente la regola di inferenza:
 - (i) Valutiamo l'espressione $2 + 9$, il cui valore è 11.
 - (ii) Sostituiamo 11 al posto di ogni occorrenza libera di y nell'espressione $9 == y$, ottenendo l'espressione $9 == 11$.
 - (iii) Valutiamo l'ultima espressione: siccome $9 \neq 11$ il risultato dell'espressione è `false`.

Segue quindi che il codice iniziale ha valore `false`.

1.3 FUNZIONI

Come in ogni linguaggio funzionale in OCaml le funzioni sono elementi del primo ordine: sono dunque espressioni, esattamente come ogni costrutto del linguaggio.

SINTASSI La sintassi per dichiarare una funzione è la seguente:

```
let <fun_name> (<par_1> : <type_1>) ... (<par_n> : type_n) :
  ↪ ret_type =
  <fun_body>.
```

Per fare un esempio:

```
let f (x : int) : int =
  let y = x * 10
  in y * y;;
```

Una funzione è quindi formata da

- un nome di funzione, in questo caso `f`;
- una lista di parametri, in questo caso il singolo parametro `x`. Notiamo che abbiamo esplicitato il tipo di `x` tramite la notazione `(x : int)`, anche se può essere sottointeso;
- un tipo di ritorno, in questo caso `int`, anch'esso opzionale;
- un corpo di funzione, in questo caso dato dal `let` interno.

INFERENZA DI TIPI L'inferenza dei tipi dell'interprete OCaml ci permette di dichiarare funzioni senza esplicitare i tipi degli argomenti e/o del risultato, come nel seguente caso:

```
let f x = x + 3;;
```

Siccome l'operatore di somma è specifico per il tipo `int` sia l'argomento della funzione che il suo risultato dovranno essere di tipo intero: il tipo di questa funzione sarà quindi denotato con `int -> int`.

CHIAMATA DI FUNZIONE Per chiamare una funzione non è necessario usare le parentesi: se dichiarassi la funzione `plus` in questo modo

```
let plus x y = x + y;;
```

l'espressione `plus 2 3` sarebbe valutata a `5`.

Possiamo anche dichiarare delle funzioni interne al corpo di un'espressione `let`:

```
let square x = x*x
  in square 3 + square 4;;
```

Questa espressione ha valore `25` poiché l'interprete calcola i valori di `square 3` e `square 4` utilizzando la formula data dal `let`, ovvero `let square x = x * x`.

FUNZIONE RICORSIVA Per dichiarare funzioni ricorsive bisogna usare la parola chiave `rec`. Ad esempio la funzione fattoriale può essere implementata in questo modo:

```
let rec fact x =
  if x = 0
  then 1
  else x * fact (x-1);;
```

Ancora una volta il meccanismo di inferenza dei tipi assegna alla funzione `fact` il tipo `int -> int`.

CURRYING Consideriamo nuovamente la funzione `plus` definita sopra. Se chiedessimo all'interprete OCaml di ricavarne il tipo, la risposta sarebbe che la funzione `plus` ha tipo `int -> int -> int`, ovvero la funzione prende un intero e restituisce un'altra funzione `int -> int`. Possiamo quindi sfruttare questo fatto per ottenere una funzione applicata parzialmente:

```
let incr = plus 1;;
```

Il tipo della funzione `incr` è `int -> int` (ovvero prende un intero e restituisce un intero), e semplicemente questa funzione si comporta come una `plus` con il primo parametro fissato ad `1`.

FUNZIONI ANONIME Possiamo dichiarare funzioni con una sintassi diversa usando la parola chiave `fun` oppure `function`.

```
let f =
  function x y = x * y;;
```

La funzione `f` è semplicemente una funzione che prende due parametri interi (chiamati `x` e `y`) e ne restituisce il prodotto, dunque è equivalente a

```
let f x y = x * y;;
```

Regola di valutazione di una funzione

La regola formale di valutazione di una chiamata di funzione è la seguente:

$$\frac{\text{Eval}(f) = \text{fun } x_1 \dots x_n = e \quad (\forall i : \text{Eval}(e_i) = v_i) \quad \text{subst}(e, x_1, \dots, x_n, v_1, \dots, v_n) = e' \quad \text{Eval}(e') = v}{\text{Eval}(f \ e_1 \dots e_n) = v}$$

La regola ci dice quindi che la chiamata di $f \ e_1 \dots e_n$ viene valutata ad un valore v se:

- f viene valutata ad una funzione (**fun**) che prende n parametri ($x_1 \dots x_n$) e restituisce un'espressione che dipende da questi parametri (e);
- i parametri attuali $e_1 \dots e_n$ vengono valutati a dei valori v_1, \dots, v_n ;
- sostituendo all'interno dell'espressione e ogni parametro formale x_i con il suo valore v_i si ottiene una nuova espressione e' ;
- la valutazione di e' dà come risultato v .

Esempio 1.3.1. Consideriamo la funzione `plus` definita sopra e la chiamata `plus 5 4`. Allora:

- `plus` viene valutata ad una funzione che prende due parametri e ne restituisce la somma, ovvero:

$$\text{Eval}(\text{plus}) = \text{fun } x \ y = x + y.$$

- I parametri vengono valutati rispettivamente a 5 e a 4.
- Sostituendo i valori 5 e 4 nell'espressione $x + y$ (al posto di x e y rispettivamente) otteniamo l'espressione `5 + 4`.
- Valutando quest'ultima espressione otteniamo 9, cioè il valore restituito dalla funzione.

FUNZIONI DI ORDINE SUPERIORE Siccome le funzioni in OCaml sono oggetti del primo ordine possiamo usarle come parametri di altre funzioni, come in questo esempio:

```
let compose (f : int -> int) (g : int -> int) (x : int) : int =
  g(f x);;
```

La funzione `compose` prende tre parametri:

- una *funzione* f da interi in interi;
- una *funzione* g da interi in interi;
- un valore intero x .

Il risultato della funzione `compose` è la composizione matematica delle funzioni f e g .

Un altro possibile esempio è il seguente:

```
let rec n_times (f : int -> int) (n : int) : (int -> int) =
  if n = 0 then (fun x -> x)
  else compose f (n_times f (n-1));;
```

Questa funzione restituisce la composizione di una funzione f con se stessa per n volte usando la ricorsione.

POLIMORFISMO Se chiedessimo all'interprete di inferire il tipo della funzione `compose` definita sopra, rimuovendo le annotazioni di tipo, la risposta sarebbe:

```
compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Le variabili `'a`, `'b` e `'c` sono *variabili di tipo*: ciò significa che la funzione `compose` non accetta solo parametri di tipo `int`, ma accetta tipi qualsiasi fintanto che tutti i tipi legati ad `'a` siano uguali, tutti i tipi legati a `'b` siano uguali e stessa cosa anche per `'c`.

1.4 LISTE

Una lista in OCaml è una collezione di valori dello stesso tipo. Le seguenti sono tutte liste:

```
let l1 = [1; 2; 3];;
let l2 = 1 :: 2 :: 3 :: [];;
let l3 = [];;
let l4 = 'a' :: 'b' :: ['c', 'd'];;
```

Il simbolo `[]` rappresenta una lista vuota, pertanto la lista `l3` ha tipo polimorfo `'a list`, mentre le altre liste hanno tipo `int list` oppure `char list`.

L'operatore `::` (chiamato *cons*) ci permette di aggiungere un elemento in testa ad una lista (a patto che il tipo dell'elemento sia uguale al tipo della lista). In realtà la sintassi con le parentesi quadre (ad esempio nel caso della prima lista) è del tutto equivalente alla sintassi con l'operatore `cons` (come nel caso della seconda lista), dunque `l1 = l2`.

Pattern matching

Per scrivere comodamente funzioni su liste è possibile usare il *pattern matching*:

```
let empty lst =
  match lst with
  | [] -> true
  | x::xs -> false;;
```

L'espressione `match lst with` ci permette di confrontare il parametro `lst` con alcuni "pattern", in modo da poter dare la risposta a seconda del pattern a cui la lista viene associata. Quindi se `lst` viene associata alla lista vuota `[]` significa che `lst` è vuota, dunque la funzione `empty` restituisce `true`; invece se `lst` viene associata ad una lista della forma `x::xs` significa che `lst` contiene almeno un elemento (cioè `x`), dunque non è vuota e pertanto restituiamo `false`.

Osserviamo che

- il pattern `[]` viene associato soltanto ad una lista vuota;
- il pattern `x::xs` viene associato ad una lista con *almeno* un elemento (`xs` rappresenta il resto della lista, che può essere vuoto oppure contenere altri elementi);
- il pattern `x::[]` viene associato ad una lista con *esattamente* un elemento;

- il pattern `x::y::xs` viene associato ad una lista con *almeno* due elementi;
- il pattern `x::y::[]` viene associato ad una lista con *esattamente* due elementi;

e così via.

Option types

Alcune funzioni su liste non possono essere applicate ad ogni tipo di lista, ma solo a liste non vuote. Un esempio è la funzione che calcola il massimo di una lista:

```
let rec max_list lst =
  match lst with
  | [] -> ???
  | [x] -> x
  | x::xs -> max x (max_list xs);;
```

Per risolvere questi problemi (senza ricorrere all'uso del `null`) OCaml implementa gli *option types*:

```
let rec max_list lst =
  match lst with
  | [] -> None
  | x::xs -> match (max_list xs) with
    | None -> x
    | Some v -> Some (max x v);;
```

Studiamo più nel dettaglio il funzionamento di `max_list`:

- se `lst` è `[]` allora il risultato è un valore particolare, chiamato **None** per indicare l'assenza di valore;
- se `lst` ha un elemento in testa `x` e una coda `xs`, allora la funzione
 - calcola ricorsivamente il valore della coda e lo usa come parametro del pattern matching;
 - se il risultato di `max_list xs` è **None**, allora la lista `xs` è vuota e il valore massimo della lista `x::xs` è `x`;
 - altrimenti se il valore massimo della lista `xs` è **Some** `v`, allora il valore massimo della lista `x::xs` è il massimo tra `x` e `v`.

Notiamo che se il risultato non è **None** allora deve essere racchiuso dal costruttore di tipo **Some**: infatti la funzione deve restituire un valore di un tipo preciso e non può restituire talvolta *option types* e talvolta tipi standard.

Il tipo di questa funzione è quindi `max_list : 'a list -> 'a option`, dove *option* indica il fatto che possiamo restituire **None**.

1.5 NUOVI TIPI DI DATO

OCaml ci permette di dichiarare nuovi tipi di dato tramite la parola chiave **type**:

```
type day =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
```

```
| Friday
| Saturday
| Sunday;;
```

Questo costrutto dichiara un nuovo tipo (`day`) che può assumere i valori `Monday`, `Tuesday`, ..., `Sunday`.

Per dichiarare funzioni che operano su valori di tipo `day` possiamo usare il pattern matching:

```
let string_of_day d =
  match d with
  | Monday    -> "Monday"
  | Tuesday   -> "Tuesday"
  | Wednesday -> "Wednesday"
  | Thursday  -> "Thursday"
  | Friday    -> "Friday"
  | Saturday  -> "Saturday"
  | Sunday    -> "Sunday";;
```

I nuovi tipi di dato possono anche "trasportare valori" di tipi standard:

```
type foo =
  | Nothing
  | Int of int
  | Pair of int * int
  | String of string;;
```

Dunque i seguenti sono tutti valori di tipo `foo`:

```
Nothing;;
Int 3;;
Pair (2, 7);;
String "ciao";;
```

Il sistema di creazione di tipi di OCaml è molto potente in quanto ci consente di creare tipi ricorsivi:

```
type bool_expr =
  | True
  | False
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
  | Not of bool_expr;;
```

Il tipo `bool_expr` è quindi un tipo usato per rappresentare espressioni booleane con gli operatori `And`, `Or` e `Not`: ad esempio i seguenti valori sono di tipo `bool_expr`.

```
True;;
And (False, True);;
Or (Not (And (True, True)), Or (Not True, False));;
```

Scriviamo ora una funzione per valutare un'espressione booleana di tipo `bool_expr`:

```
let rec bool_eval expr =
  match expr with
  | True      -> true
  | False     -> false
  | Or (e1, e2) -> bool_eval e1 || bool_eval e2
  | And (e1, e2) -> bool_eval e1 && bool_eval e2
  | Not e1      -> not (bool_eval e1);;
```

2 | SINTASSI E SEMANTICA

2.1 SINTASSI

Per descrivere un linguaggio di programmazione abbiamo bisogno di diversi strumenti:

- una **grammatica libera dal contesto** per descrivere la sintassi;
- un metodo per descrivere le regole di scoping e il sistema dei tipi, chiamato **semantica statica**;
- un metodo per descrivere i comportamenti del linguaggio, detto **semantica dinamica**.

Per analizzare la sintassi di un linguaggio abbiamo bisogno quindi di studiarne la grammatica: tuttavia questo può essere scomodo in alcune circostanze, in quanto possono presentarsi problemi di ambiguità tra espressioni.

Si ricorre quindi all'uso dei cosiddetti **Alberi di Sintassi Astratta** (abbreviati in AST, dall'inglese Abstract Syntax Tree): ogni nodo dell'albero rappresenta un'operazione e i suoi nodi figli rappresentano gli operandi dell'operazione (e possono essere a loro volta altre operazioni).

2.2 MECCANISMI DI INFERENZA

Per asserire proprietà (statiche o dinamiche) dei nostri programmi abbiamo bisogno di un meccanismo di inferenza.

Definizione 2.2.1 **judgment.** Si dice **judgment** (o anche **sentenza**) un enunciato che asserisca una proprietà di un oggetto.

Ad esempio, la frase "Il numero 3 è dispari" è un judgment.

Definizione 2.2.2 **Regola di inferenza.** Siano J_1, \dots, J_n, J delle sentenze. Una **regola di inferenza** è un'implicazione della forma

$$\frac{J_1 \dots J_n}{J},$$

il cui significato è che se J_1, \dots, J_n sono sentenze derivabili dal sistema assiomatico, allora lo è anche J .

I judgment J_1, \dots, J_n vengono detti *premesse* o *precondizioni*, mentre il judgment J viene detto *conclusione* o *postcondizione*.

Una regola di inferenza senza premesse si dice *assioma*.

Ad esempio la seguente regola è un assioma:

$$\overline{0 : \text{nat}},$$

mentre la seguente è una regola con premesse:

$$\frac{n : \text{nat}}{s(n) : \text{nat}},$$

dove con $s(n)$ intendiamo il successore di n .

La strategia che useremo per dimostrare sentenze della forma $s : A$ (il cui significato è "l'oggetto s soddisfa la proprietà A ") è la seguente:

- troviamo una regola R la cui conclusione corrisponde alla sentenza $s : A$;
- dimostriamo tutte le precondizioni con la stessa strategia.

2.3 SEMANTICA STATICA

La semantica statica è il meccanismo che ci permette di descrivere proprietà di un programma che si manifestano senza doverlo eseguire. Queste proprietà sono spesso controllate dal compilatore o da strumenti esterni e ci permettono di avere un controllo statico sui programmi che vogliamo eseguire.

Per mostrare l'uso della semantica statica studiamo un semplice linguaggio di programmazione che può solo fare calcoli. La grammatica di questo linguaggio è la seguente:

```
E ::= Const | Ide | (Times E1 E2) | (Plus E1 E2) | (Let Ide E1 E2)
Const ::= 0 | 1 | 2 | ...
Ide ::= x | y | z | ...
```

Consideriamo il seguente judgment: $E : \text{ok}$ se tutti gli identificatori contenuti in E sono definiti correttamente tramite un'espressione di tipo `Let`.

Le regole per le costanti e per le operazioni di addizione e moltiplicazione sono molto semplici:

$$\frac{}{\text{Const} : \text{ok}} \quad \frac{E1 : \text{ok} \quad E2 : \text{ok}}{(\text{Times } E1 \ E2) : \text{ok}} \quad \frac{E1 : \text{ok} \quad E2 : \text{ok}}{(\text{Plus } E1 \ E2) : \text{ok}}$$

Tuttavia non possiamo al momento descrivere una regola per verificare la correttezza delle singole variabili e delle espressioni `Let`: data un'espressione del tipo `Ide` non abbiamo abbastanza informazione per decidere se essa è corretta oppure no, poiché potrebbe essere sia libera nell'espressione generale (e in tal caso sarebbe sbagliata), sia legata da qualche `Let` precedente.

Abbiamo quindi bisogno di introdurre una struttura di supporto per recuperare le informazioni relative agli identificatori. Chiameremo questa struttura **tabella dei simboli**, ed essa conterrà tutti i nomi delle variabili che abbiamo dichiarato nel programma. Inoltre se Γ è una tabella dei simboli, il judgment $\Gamma \vdash e : A$ significa che considerando l'*ambiente* Γ l'espressione e ha la proprietà A .

Possiamo quindi esprimere completamente la condizione di correttezza di un programma del nostro linguaggio:

$$\frac{}{\Gamma \vdash \text{Const} : \text{ok}} \quad \frac{\Gamma \vdash E1 : \text{ok} \quad \Gamma \vdash E2 : \text{ok}}{\Gamma \vdash (\text{Times } E1 \ E2) : \text{ok}} \quad \frac{\Gamma \vdash E1 : \text{ok} \quad \Gamma \vdash E2 : \text{ok}}{\Gamma \vdash (\text{Plus } E1 \ E2) : \text{ok}} \quad \frac{x \in \Gamma}{\Gamma \vdash x : \text{ok}} \quad \frac{\Gamma \vdash E1 : \text{ok} \quad \Gamma \cup \{x\} \vdash E2 : \text{ok}}{\Gamma \vdash (\text{Let } x \ E1 \ E2) : \text{ok}}$$

Simulazione in OCaml

Cerchiamo ora di simulare le regole dell'analisi statica in OCaml. Introduciamo innanzitutto le definizioni di tipo per le espressioni e gli identificatori:

```
type ide = string;;
type expr =
| Int    of int
| Den    of ide * int
| Plus   of expr * expr
| Times  of expr * expr
| Let    of ide * expr * expr;;
```

Dato un ambiente e un identificatore, controlliamo se l'identificatore esiste nell'ambiente:

```
let rec lookup st x =
  match st with
  | []    -> false
  | y::ys -> if x = y then true
              else lookup ys x;;
```

La funzione che controlla se l'espressione è ben formata è quindi la seguente:

```
let rec check (e : expr) (st : string list) =
  match e with
  | Int n -> true
  | Den id -> lookup st x
  | Plus (e1, e2) -> (check e1 st) && (check e2 st)
  | Times (e1, e2) -> (check e1 st) && (check e2 st)
  | Let (id, e1, e2) -> (check e1 st) && (check e2 (x::st))
```

2.4 SEMANTICA DINAMICA

Vogliamo ora studiare la **semantica dinamica**, che ci dà le regole di esecuzione di un programma e ci permette di calcolarne il risultato. Vi sono diversi stili di semantica (in particolare *denotazionale*, *operazionale* e *assiomatica*), ma noi ci concentreremo sulla semantica operazionale ed in particolare su due tipi particolari:

- la **Structural Operational Semantics**, anche chiamata *semantica Small Steps*, che descrive ogni passo dell'esecuzione di un programma;
- la **Natural Semantics**, che descrive il risultato dell'esecuzione di un programma completo.

Semantica Small Steps

L'idea di base della semantica small steps è che ogni passo di valutazione ci porta da un'espressione ad un'espressione più semplice, fino ad arrivare ad un'espressione che non può più essere semplificata (un valore).

Si dice quindi **sistema di transizione** una tupla (S, I, F, \rightarrow) dove:

- S è l'insieme dei possibili stati della macchina astratta del linguaggio;
- I è l'insieme degli stati iniziali;
- F è l'insieme degli stati finali;

- $\rightarrow \subseteq S \times S$ è la *relazione di transizione*, che descrive l'effetto di un singolo passo di valutazione.

Nel caso del nostro semplice linguaggio possiamo definire:

- S come l'insieme delle espressioni aritmetiche sintatticamente corrette (in qualche ambiente), ovvero

$$S := \{ e : \exists \Gamma \text{ tale che } \Gamma \vdash e : \text{ok} \};$$

- I come l'insieme delle espressioni aritmetiche "chiuse", ovvero che non contengono variabili libere:

$$I := \{ e : \emptyset \vdash (e : \text{ok}) \};$$

- F come l'insieme dei valori interi:

$$F := \{ \text{Int } n : n \in \mathbb{N} \}.$$

Iniziamo a studiare le regole di valutazione.

TIMES Facciamo l'esempio della regola del costrutto Times:

$$\frac{\frac{\frac{}{(\text{Times } (\text{Int } n) (\text{Int } m)) \rightarrow \text{Int } (n * m)}}{e1 \rightarrow e1'} (\text{Times } e1 e2) \rightarrow (\text{Times } e1' e2)}{e2 \rightarrow e2'} (\text{Times } (\text{Int } n) e2) \rightarrow (\text{Times } (\text{Int } n) e2')$$

Questa regola composta ci dice che

- se stiamo cercando di calcolare il risultato di Times applicato a due valori, il risultato è un valore (in particolare è $\text{Int } (n * m)$);
- se entrambi gli operandi non sono valori, eseguiamo uno step sul primo operando (che quindi può diventare un valore, ma può anche dover essere semplificato ancora) e rivalutiamo l'espressione;
- se il primo operando è un valore ma il secondo non lo è, eseguiamo uno step sul secondo operando e rivalutiamo l'espressione.

Facciamo alcune osservazioni:

- questa regola di moltiplicazione è **eager**: prima valuta completamente entrambi gli operandi, poi valuta l'espressione completa;
- inoltre anche l'ordine in cui vengono eseguite le valutazioni è evidente: prima viene eseguita la valutazione del primo operando, poi viene eseguita la valutazione del secondo.

LET Studiamo ora la regola di valutazione del Let:

$$\frac{\frac{}{(\text{Let } x (\text{Int } n) e2) \rightarrow e2[x := (\text{Int } n)]}}{e1 \rightarrow e1'} (\text{Let } x e1 e2) \rightarrow (\text{Let } x e1' e2)$$

Questa regola ci dice che

- se abbiamo valutato completamente $e1$ al valore $(\text{Int } n)$ allora lo step consiste nel restituire l'espressione $e2$ dove tutte le occorrenze di x vengono sostituite da occorrenze di $(\text{Int } n)$;
- altrimenti eseguiamo uno step di valutazione sull'espressione $e1$ e rivalutiamo l'espressione.

Semantica naturale

Nel caso della semantica naturale non siamo più interessati a semplificare un'espressione *un passo alla volta*, ma vogliamo valutare in un passo solo un'intera espressione. Per far ciò avremmo bisogno di

- un insieme E di espressioni valutabili,
- un insieme V di valori (che non è necessariamente un sottoinsieme di E),
- una relazione $\Rightarrow \subseteq E \times V$ (anche indicata con \Downarrow), detta *relazione di valutazione*.

Ad esempio nel caso del nostro semplice linguaggio possiamo dare una semantica naturale nel seguente modo:

- E è l'insieme di tutte le espressioni ben formate, ovvero

$$E = \{e : \Gamma \vdash (e : \text{ok})\};$$

- V è l'insieme dei valori interi,
- alcune regole di valutazione sono le seguenti:

$$\frac{}{\text{Int } n \Rightarrow \text{Int } n}$$

$$\frac{e1 \Rightarrow \text{Int } n \quad e2 \Rightarrow \text{Int } m}{\text{Times } e1 \ e2 \Rightarrow \text{Int } (n*m)}$$

$$\frac{e1 \Rightarrow (\text{Int } n) \quad e2[x := (\text{Int } n)] \Rightarrow (\text{Int } m)}{(\text{Let } x \ e1 \ e2) \Rightarrow \text{Int } m}$$

Osserviamo che la regola data sopra per il `Let` è una regola di valutazione *eager*, nel senso che valuta tutti gli operandi prima di passare alla valutazione dell'operazione principale. Una regola di valutazione *lazy* per il `Let` è ad esempio la seguente:

$$\frac{e2[x := e1] \Rightarrow (\text{Int } m)}{\text{Let } x \ e1 \ e2 \Rightarrow (\text{Int } m)}.$$

Tuttavia per definire correttamente un interprete abbiamo bisogno di un metodo per recuperare a tempo di esecuzione le informazioni relative alle variabili legate dai `Let` (e successivamente dalle funzioni): definiamo quindi il concetto di *ambiente* e di *binding*.

Si dice **binding** un'associazione tra un nome (di variabile) e un valore; si dice **ambiente** una funzione

$$\text{env} : \text{Ide} \rightarrow \text{Values} \cup \{\text{Unbound}\}$$

dove Ide è l'insieme degli identificatori di variabili, Values è l'insieme dei valori e Unbound è il valore particolare assunto da una variabile che non è legata ad alcun valore. Per aggiungere un nuovo binding ad un ambiente useremo la notazione $\text{env } [x := v]$, che indica la funzione che si comporta esattamente come env per ogni $y \in \text{Ide} \setminus \{x\}$, mentre $\text{env}(x) = v$.

Una possibile prima implementazione di questo tipo di ambiente in OCaml è la seguente:

```
let empty_env = [];;

let rec lookup env x =
  match env with
  | [] -> failwith ("not found")
  | (a, v)::_ as _ -> if x = a then v
```

```

else lookup as x;;

let bind x v env = (x, v)::env;;

```

Possiamo quindi specificare meglio la nostra costruzione della semantica naturale: abbiamo bisogno di

- un insieme di stati

$$S = \{ \rho \triangleright e : \rho \in \text{Env}, e \in \text{Exp}, \emptyset \vdash (e : \text{ok}) \}.$$

Ogni stato rappresenta una coppia formata da un ambiente e da un'espressione da valutare in quell'ambiente.

- un insieme di valori, che nel nostro caso sono i numeri naturali,
- una relazione di valutazione $\Rightarrow \subseteq S \times V$.

Le regole di valutazione dell'interprete diventano quindi ad esempio:

$$\begin{array}{c}
\frac{\text{env}(\text{id}) = v \in V}{\text{env} \triangleright (\text{Den id}) \Rightarrow v} \\
\\
\frac{\text{env} \triangleright e_1 \Rightarrow v_1 \quad \text{env} \triangleright e_2 \Rightarrow v_2 \quad v_1 \cdot v_2 = v}{\text{env} \triangleright (\text{Times } e_1 \ e_2) \Rightarrow v} \\
\\
\frac{\text{env} \triangleright e \Rightarrow v' \quad \text{env } [x := v'] \triangleright \text{body} \Rightarrow v}{\text{env} \triangleright (\text{let } x = e \text{ in body}) \Rightarrow v}
\end{array}$$

Una possibile ottimizzazione di questo processo viene data dall'uso degli **indici di De Bruijn**: a tempo di compilazione sostituiamo il nome di ogni variabile con il numero di Let che bisogna attraversare per raggiungere il Let in cui è stata definita tale variabile. Quindi ad esempio l'espressione

Let ("x", (Int 5), (Let ("z", (Int 17), (Times (Den "z", Den "x")))))

può essere trasformata nell'espressione

Let ((Int 5), (Let ((Int 17), (Times (Den 0, Den 1)))))

2.5 TIPI DI DATO

Finora il nostro semplice linguaggio può produrre soltanto valori di tipo intero, quindi non vi è la necessità di un sistema di tipi. Tuttavia quando i valori cominciano ad essere più complicati, un **type system** può essere utile per diversi motivi:

- i tipi sono utili a livello di *progetto*: organizzano l'informazione e ci consentono di astrarre esplicitamente sui dati;
- sono utili a livello di *programma*: identificano e prevengono alcuni errori automaticamente;
- sono anche utili a livello di *implementazione*: tipi diversi richiedono risorse diverse (ad esempio un booleano richiede solo un bit, mentre un valore floating-point ad alta precisione ne richiede 64) e quindi forniscono alcune informazioni necessarie alla macchina astratta per allocare lo spazio di memoria.

Distinguiamo innanzitutto tra diversi tipi di dati:

- un dato si dice *denotabile* se può essere associato ad un nome;

- un dato si dice *esprimibile* se può essere il risultato della valutazione di un'espressione;
- un dato si dice *memorizzabile* se può essere memorizzato in una variabile.

Ad esempio le funzioni in OCaml sono denotabili ed esprimibili ma non sono memorizzabili, mentre in C sono solamente denotabili.

Descrittori di dato

Quando studiamo i tipi di dato vogliamo studiare sia la loro semantica, sia la loro implementazione. Partendo da quest'ultima, sembra necessario che un tipo di dato nella sua rappresentazione concreta contenga una "descrizione del tipo": a run-time vogliamo infatti (ad esempio) essere certi che il tipo del dato che abbiamo sia quello previsto dall'operazione che stiamo effettuando.

In OCaml potremmo rappresentare questo fatto nel seguente modo:

```
(* Espressione sintattica *)
type exp =
  | EInt of int
  | EBool of bool

(* Tipo a run-time *)
type evT =
  | Int of int
  | Bool of bool

(* Funzione per il typechecking *)
let typecheck descr x =
  match descr with
  | "int" -> (match x with
              | Int n -> true
              | _      -> false)
  | "bool" -> (match x with
                | Bool b -> true
                | _      -> false)
```

Tuttavia l'uso dei descrittori di dato può essere superfluo a seconda del tipo di linguaggio considerato.

- Se l'informazione sui tipi è conosciuta completamente a tempo di compilazione (come nel caso di OCaml) si possono eliminare i descrittori di dato, in quanto il typecheck è effettuato dal compilatore (*typecheck statico*).
- Se l'informazione sui tipi è conosciuta solamente a tempo di esecuzione (come ad esempio in Javascript) i descrittori sono necessari per tutti i tipi e il typechecking è completamente *dinamico*.
- Se l'informazione sui tipi è conosciuta parzialmente a tempo di compilazione (come nel caso di Java) i descrittori di dato devono contenere solo l'informazione "dinamica" e il typecheck è effettuato parzialmente a tempo di compilazione e parzialmente a tempo di esecuzione.

I tipi predefiniti vengono anche chiamati *tipi scalari*. Tra essi vi sono:

- gli interi,
- i booleani,
- i caratteri,

- i numeri reali (floating point),
- il tipo **void**, rappresentato in OCaml da `()`, con le seguenti caratteristiche:
 - ha un solo valore,
 - non ha operazioni,
 - serve per implementare operazioni che modificano lo stato senza restituire un vero valore.

Tuttavia i linguaggi definiscono anche dei **tipi composti**. Ve ne sono diversi:

- i record (chiamati **struct** in C),
- i record varianti, oppure *sum types* (in cui solo un tipo è attivo in un dato istante di tempo, chiamati **union** in C),
- gli array,
- gli insiemi,
- i puntatori.

Record

I record sono stati definiti per manipolare in modo unitario dati di tipo eterogeneo. Ad esempio il tipo

```
struct studente {
    int matricola;
    char nome[20];
}
```

definisce un tipo di dato che rappresenta uno studente, con il suo nome e numero di matricola.

L'implementazione di un record viene fatta *sequenzialmente*: i vari campi occupano posti consecutivi in memoria. In particolare abbiamo due possibili strategie:

- allineamento alla parola;
- packed record.

Nel caso di *allineamento alla parola* ogni campo del record deve occupare una o più parole intere, dunque se occupa meno spazio di una parola (tipicamente 4 byte) allora viene lasciato dello spazio libero (chiamato *padding*). Questo porta ad uno spreco di memoria, ma contemporaneamente permette di avere degli accessi molto semplici ai campi (si trovano tutti all'inizio di una parola di memoria).

Nel caso di *packed record* i campi vengono scritti consecutivamente, senza lasciare spazio tra un campo e l'altro e quindi senza rispettare l'allineamento alla parola. L'accesso ai campi si fa più complicato, tuttavia non vi è spreco di memoria.

Simulazione dei record in OCaml

Per simulare il comportamento dei record, estendiamo la sintassi astratta del nostro linguaggio con i seguenti costruttori:

```

type label = Lab of string
type expr = ...
| Record of (label * expr) list
| Select of label * expr

```

Possiamo quindi dichiarare record della seguente forma:

```
Record [(Lab "size", Int 5), (Lab "weight", Int 255)]
```

Per interpretare la creazione di un record e l'operazione di selezione dobbiamo innanzitutto estendere i tipi esprimibili a run-time dal linguaggio:

```

type evT = ...
| RecordEv of (label * evT) list

```

A questo punto introduciamo una funzione trovare un valore in un **RecordEv** data una label:

```

let rec lookupRecord (body : (label * evT) list) (lab : label) :
  ↪ evT =
  match body with
  | [] -> raise FieldNotFound
  | (lab', value)::rs -> if lab' = lab then value
                        else lookupRecord rs lab

```

Possiamo quindi estendere la funzione di valutazione con la valutazione dei due nuovi campi:

```

let rec eval (exp : expr) : evT =
  match exp with
  | ...
  | Record recordBody -> RecordEv (evalRecord recordBody)
  | Select(e, lab) -> (match eval e with
                       | RecordEv body -> lookupRecord body
                           ↪ lab
                       | _ -> raise TypeMismatch)
and evalRecord (body : (label * expr) list) : (label * evT) list =
  match body with
  | [] -> []
  | (lab, e)::rs -> (lab, eval e)::evalRecord rs

```

Array

Un **array** è una collezione di dati omogenei. In particolare, astrattamente un array è una funzione da un insieme di indici (solitamente di tipo intero) ad un insieme di elementi (di tipo qualunque, ma solitamente non funzionale).

La principale operazione ammessa sugli array è la selezione di un elemento: la modifica infatti non è propriamente un'operazione sull'array ma piuttosto un'operazione sulla locazione di memoria che memorizza quel dato.

Gli array vengono memorizzati in locazioni continue di memoria: nel caso degli array multidimensionali (le matrici) si può sfruttare sia un *ordine di riga* in cui vengono memorizzate le righe in ordine, oppure un *ordine di colonna*; solitamente il primo è più usato.

Per accedere agli elementi di un array si può usare una semplice formula: dato l'indirizzo della base b dell'array e un offset i , per selezionare l'elemento i -esimo basta calcolare

$$b + c \cdot i,$$

dove c è la dimensione in byte degli elementi contenuti nell'array. Esiste una formula analoga (ma più complicata) per gli array multidimensionali.

Puntatori

I **puntatori** sono dei riferimenti a delle locazioni di memoria, oppure la costante `NULL`.

2.6 TYPE-SYSTEM

Cerchiamo ora di comprendere come deve essere implementato un *type-system* per un linguaggio di programmazione.

Un **type system** è una funzione che associa ad ogni oggetto un *tipo*. Esaminando il flusso dei valori calcolati, il sistema ci assicura che il programma non contenga errori di tipo.

Per tenere traccia del tipo di ogni identificatore nel nostro programma dobbiamo usare un **ambiente dei tipi**. Indicheremo con

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2, \dots, x_n : \tau_n$$

l'ambiente dei tipi che associa all'espressione x_i il valore τ_i . Inoltre useremo il simbolo

$$(\Gamma, x : \tau)$$

per indicare l'ambiente dei tipi che associa x al tipo τ e associa ogni espressione y diversa da x il tipo ad essa associato precedentemente da Γ . Infine, il judgment

$$\Gamma \vdash x : \tau$$

indica che nell'ambiente dei tipi Γ l'espressione x ha tipo τ .

Definiamo ora un semplice sistema dei tipi per le nostre espressioni.

$$\begin{array}{c} \Gamma \vdash n : \text{int} \\ \Gamma \vdash b : \text{bool} \\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \\ \frac{\Gamma \vdash e1 : \tau' \quad (\Gamma, x : \tau') \vdash e2 : \tau}{\Gamma \vdash \text{let } x = e1 \text{ in } e2 : \tau} \\ \frac{\Gamma \vdash e1 : \tau_1 \quad \Gamma \vdash e2 : \tau_2 \quad \Gamma \vdash \otimes : \tau_1 \times \tau_2 \rightarrow \tau}{\Gamma \vdash e1 \otimes e2 : \tau} \\ \frac{\Gamma \vdash \text{cond} : \text{bool} \quad \Gamma \vdash e1 : \tau \quad \Gamma \vdash e2 : \tau}{\Gamma \vdash (\text{if cond then } e1 \text{ else } e2) : \tau} \end{array}$$

Nel caso delle funzioni il type system deve diventare necessariamente più complicato. Definiamo il tipo di una funzione come il *tipo freccia* $\tau_1 \rightarrow \tau_2$, dove τ_1, τ_2 sono due tipi del nostro type system.

Abbiamo quindi le seguenti due regole di inferenza, una per le definizioni di funzione e una per le applicazioni:

$$\begin{array}{c} \frac{(\Gamma, x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \text{fun } (x : \tau_1) = e : \tau_1 \rightarrow \tau_2} \\ \frac{\Gamma \vdash e1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e2 : \tau_1}{\Gamma \vdash \text{App}(e1, e2) : \tau_2} \end{array}$$

Queste due regole possono tuttavia risultare insufficienti nel caso di *scoping dinamico*. Parleremo in seguito del concetto di scoping statico e dinamico, ma per il momento accontentiamoci di questo esempio.

```
let x = 10 in
let f = fun k -> k + x in
let x = 5 in
  f 10;;
```

Se analizziamo questo codice abbiamo due possibili interpretazioni:

- la x contenuta nella chiamata a funzione f x si riferisce alla variabile x legata al valore 10 dal primo operatore **let**: infatti essa è l'unica variabile chiamata x al momento della definizione della funzione;
- la x contenuta nella chiamata a funzione f x si riferisce alla variabile x legata al valore 5 dal terzo operatore **let**.

Nel primo caso (chiamato *scoping statico*) il valore dell'espressione complessiva è 20; nel secondo (chiamato *scoping dinamico*) il valore è invece 15. Tuttavia nel secondo caso le regole del nostro type-system non possono essere controllate staticamente: l'espressione

```
let x = 10 in
let f = fun k -> k + x in
let x = false in
  f 10;;
```

conterrebbe un ovvio errore di tipo che non si manifesterebbe fino al momento dell'esecuzione.

2.6.1 Implementazione in OCaml di un type system

Innanzitutto dobbiamo decidere quali sono i tipi del nostro linguaggio:

```
type tval =
  | TInt
  | TBool
  | FunT of tval * tval
```

A questo punto dobbiamo creare l'ambiente dei tipi:

```
type tenv = ide -> tval
```

Le funzioni `bind_t` e `empty_tenv` ci consentono rispettivamente di aggiungere un binding ad un ambiente dei tipi dato e di ottenere l'ambiente vuoto.

```
let bind_t (env : tenv) (id : ide) (v : tval) : tenv =
  fun x -> if x = id then v else env x
let empty_tenv : tenv = fun (x : ide) -> raise EmptyEnv
```

Per poter sfruttare il typechecker dobbiamo modificare il tipo delle espressioni che usiamo, aggiungendo dove necessario i tipi dei parametri.

```
type texp =
  | EInt of int
  | EBool of bool
  | Den of ide
  | Add of texp * texp
  | ...
  | Let of ide * texp * texp
  | Fun of ide * tval * texp
  | Apply of texp * texp
```

Osserviamo che l'unica modifica fatta è al costruttore delle funzioni anonime: in questo caso dobbiamo sapere di che tipo è il parametro per poter dedurre il tipo della funzione.

Il typechecker rispetta precisamente le regole semantiche date in precedenza:

```

let rec teval (e : texp) (env : tenv) : tval =
  match e with
  | EInt a -> TInt
  | EBool b -> TBool
  | Den s -> env s
  | Let (id, e1, e2) ->
    let t = teval e1 tenv in
    teval e2 (bind tenv id t)
  | Add (e1, e2) ->
    let t1 = teval e1 tenv in
    let t2 = teval e2 tenv in
    ( match t1, t2 with
      | (TInt, TInt) -> TInt
      | _ -> raise WrongType )
  | ...
  | If (cond, e1, e2) ->
    let tcond = teval cond tenv in
    ( match tcond with
      | TBool ->
        let t1 = teval e1 tenv in
        let t2 = teval e2 tenv in
        ( match t1, t2 with
          | TInt, TInt -> TInt
          | TBool, TBool -> TBool
          | _ -> raise WrongType ))
      | _ -> raise WrongType )
  | Fun (id, tpar, body) ->
    let tenv1 = bind tenv id tpar in
    let t2 = teval body tenv1 in
    FunT (t1, t2)
  | Apply (e1, e2) ->
    let f = teval e1 tenv in
    ( match f with
      | Fun(t1, t2) ->
        if (teval e2 tenv = t1) then t2
        else raise WrongType
      | _ -> raise WrongType )

```

Possiamo anche estendere il type system definito in precedenza con il supporto ad altri tipi, ad esempio con il supporto alle funzioni ricorsive, oppure alle coppie di valori, eccetera.

2.7 CHIAMATE A FUNZIONE

Nel nostro interprete non abbiamo ancora definito una sintassi e una semantica per la dichiarazione e l'applicazione di funzioni. La dichiarazione di funzione ci pone già diversi problemi:

- come descrivo il passaggio dei parametri?
- se nella funzione ci sono variabili dichiarate esternamente alla funzione (variabili *non-locali*), dove trovo il loro valore?

Per questo conviene analizzare prima un costrutto più semplice, ovvero i **blocchi**.

2.7.1 Blocchi

Un blocco è una procedura senza nome e senza parametri. Ad esempio nel C i blocchi vengono realizzati attraverso l'uso di parentesi graffe:

```
{ // starting point of the first block
    int x = 5;
    { // starting point of the second block
        int y = 2;
        int z = x + y;
    } // ending point of the second block
    x = x + 2;
} // ending point of the second block
```

Il sistema per gestire i blocchi (e, come vedremo in seguito, le funzioni) viene chiamato **stack dei record di attivazione**. Un record di attivazione è semplicemente una porzione riservata di memoria che contiene spazio sufficiente per memorizzare delle variabili, più dello spazio aggiuntivo per ulteriori dati che esamineremo più avanti.

I record di attivazione vengono gestiti tramite uno stack in quanto è la struttura dati più adatta allo scopo. Nel caso precedente, ad esempio, lo stack dei record di attivazione subisce le seguenti modifiche:

1. All'inizio lo stack *S* è vuoto.
2. Quando entriamo nel primo blocco viene creato un nuovo record di attivazione R_1 , contenente lo spazio per la variabile intera *x*; questo record viene immediatamente inserito nello stack con un'operazione di *push*.
3. All'entrata nel secondo blocco verrà creato un nuovo record di attivazione R_2 contenente lo spazio per *y* e *z*; anch'esso viene immediatamente inserito nello stack con un'operazione di *push*.
4. Nell'inizializzare *z* si usa il valore di *x* contenuto nel record R_1 : *x* è un riferimento *non-locale* nel record attuale.
5. Alla fine del blocco interno si rimuove il record di attivazione dallo stack con una *pop*. Da questo punto in poi le variabili *y* e *z* non sono più allocate.
6. Alla fine del blocco esterno si rimuove il record R_1 tramite una *pop*.

Osserviamo che dobbiamo prevedere dello spazio in ogni record per memorizzare i risultati intermedi, altrimenti non potremmo svolgere calcoli complessi. Inoltre abbiamo bisogno di un meccanismo per risalire da un record ad un altro record posizionato *più in alto* nella catena di record contenuti nello stack: questo meccanismo ci è dato dal **control link**.

Un record di attivazione per un blocco è quindi formato da 3 parti:

- un **control link** (o *puntatore di catena dinamica*) che rappresenta il puntatore dal blocco corrente al record appena precedente nello stack;
- dello spazio per le variabili di istanza;
- dello spazio per i risultati temporanei.

Ogni volta che facciamo una *push* di un record di attivazione sullo stack dobbiamo aggiornare il control link del blocco corrente per farlo puntare al blocco precedente; ogni volta che facciamo una *push* ci basterà distruggere questo puntatore. Inoltre il puntatore al record in cima allo stack viene chiamato **environment pointer** o **stack pointer** e viene aggiornato ogni volta che aggiungiamo o rimuoviamo un record dallo stack.

Regole di scope nel caso dei blocchi

Nell'esempio precedente scritto in C abbiamo notato che abbiamo bisogno di un meccanismo per calcolare il valore di riferimenti non locali. I due meccanismi vengono chiamati **scoping statico** o **scoping dinamico**:

- nel caso di scoping statico i riferimenti non locali vengono risolti nel blocco più vicino *sintatticamente* nella struttura del programma;
- nel caso di scoping dinamico i riferimenti non locali vengono risolti nel record di attivazione precedente nell'*esecuzione* del programma.

Fortunatamente nel caso dei blocchi i due metodi sono equivalenti: possiamo considerare l'uno o l'altro senza distinzioni.

2.7.2 Funzioni e procedure

Anche nel caso di funzioni e procedure si usano i record di attivazione: in questo caso il record deve contenere spazio almeno per:

- il control link;
- i parametri della funzione;
- l'indirizzo di ritorno;
- le variabili locali;
- i risultati intermedi;
- il valore restituito (anche se possiamo considerarlo come un risultato intermedio);
- indirizzo in cui inserire il valore restituito.

Passaggio dei parametri

Sappiamo che esistono diverse modalità per il passaggio dei parametri:

- nel caso del *passaggio per valore*, implementato ad esempio dal C e dal Java, nel record di attivazione viene copiato il valore del parametro: la funzione non può modificare il contenuto del parametro e non vi è *aliasing* (più variabili che si riferiscono alla stessa locazione di memoria).
- nel caso del *passaggio per riferimento* viene copiato nel record di attivazione l'*indirizzo* del parametro attuale: si crea quindi una situazione di *aliasing*, da cui segue che la funzione può modificare il contenuto della variabile.

Bisogna prestare attenzione al fatto che nel caso del C e del Java l'unica modalità di passaggio dei parametri consentita è il passaggio per valore: infatti quando si passa un puntatore/riferimento ad un oggetto in realtà si sta copiando il valore della variabile, soltanto che questo valore è una locazione di memoria.

Regole di scope nel caso di funzioni

Consideriamo ora il seguente esempio:

```
var x = 1;
function g(z) {
    return x + z;
}
```



```
function f(y) {
    x = y+1;
    return g(y*x);
}
f(3);
```

Qual è il valore di $f(3)$ nel caso di scoping statico o dinamico?

- Nel caso di scoping statico la variabile x contenuta nella funzione g si riferisce alla variabile dichiarata nella prima linea di codice. Dunque nell'esecuzione della funzione i passi di calcolo saranno:
 - nel calcolo di $f(4)$ si pone x uguale a 4 e si calcola $g(3*4)$;
 - nel calcolo di $g(12)$ si usa il valore di x definito staticamente, ovvero $x = 1$, e si restituisce quindi 13, che è quindi il valore complessivo della chiamata a funzione $f(3)$.
- Nel caso di scoping dinamico ogni riferimento non locale viene risolto *dinamicamente*, durante l'esecuzione del programma. Dunque:
 - nel calcolo di $f(4)$ si pone x uguale a 4 e si calcola $g(3*4)$;
 - nel calcolo di $g(12)$ si usa il valore di x contenuto nel record di attivazione precedente, che è il record che contiene la chiamata a $f(3)$: segue quindi che x vale 4 e il risultato della funzione è 16.

Nel caso delle chiamate a funzione quindi i due meccanismi sono diversi e portano a comportamenti diversi: i linguaggi di programmazione devono quindi scegliere se adottare lo scoping statico (scelta fatta dalla maggior parte dei linguaggi) o lo scoping dinamico (scelta fatta principalmente dal LISP e da alcuni linguaggi di scripting).

Scoping statico

Analizziamo ora più nel dettaglio il funzionamento dello scoping statico: il record di attivazione di una funzione nel caso di scoping statico deve contenere, in aggiunta a quanto detto precedentemente, uno **static link** che collega il blocco corrente al blocco padre nella gerarchia statica.

I due *link* (quello statico e quello dinamico) hanno quindi ruoli diversi: il primo viene usato per risolvere i riferimenti non locali ed è determinato staticamente, senza dover eseguire il codice, mentre il secondo è determinato solamente dall'ordine delle chiamate di funzione e quindi ha senso soltanto durante l'esecuzione del programma.

Esempio 2.7.1. Supponiamo di aver a che fare con il seguente codice simil-C:

```
{ // MAIN
  int x;
  void A() {
    x = x+1;
  }
  void B () {
    int x;
    void C (int y) {
      int x;
      x = y+2;
      A ();
    }
    x = 0;
    A();
    C(3);
  }
}
```

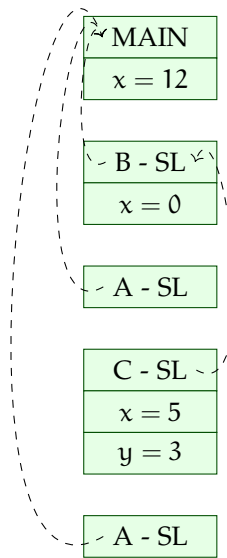


Figura 1: Struttura dei record di attivazione

```

    x = 10;
    B();
}

```

Siccome le funzioni A e B sono dichiarate nel blocco "main", ogni volta che verranno chiamate avremo che il loro *static link* punterà al blocco main; invece la funzione C avrà uno static link che punterà al blocco della funzione B dove è stata dichiarata.

Il risultato dell'esecuzione del codice è dato dalla [Figura 1](#) (le frecce tratteggiate sono i puntatori di catena statica):

- nel main x viene inizializzato a 12 e viene chiamata la funzione B();
- la funzione dichiara una nuova variabile x, che viene inizializzata a 0, dopo chiama la funzione A();
- A deve calcolare $x = x + 1$, ma siccome x è un riferimento non locale deve cercarlo seguendo il puntatore di catena statica: la x trovata è quella dichiarata nel main, che vale 12. Il programma pone quindi la x del main uguale a 13 e ritorna;
- continua l'esecuzione della funzione B, che chiama a questo punto C(3);
- la funzione C dichiara una nuova variabile x e la pone uguale a $y + 2$; siccome y è un riferimento locale il valore della x locale diventa quindi 5. Subito dopo C chiama la funzione A;
- ancora una volta A deve trovare il riferimento non locale x seguendo il puntatore di catena statica e, come nel caso precedente, trova il valore dichiarato nel main e lo aggiorna a 14;
- si conclude l'esecuzione del programma.

Funzioni come valori e chiusure

Nel caso dei linguaggi funzionali le funzioni tuttavia sono dei valori: possono essere passate ad altre funzioni e possono essere restituite come risultato di funzioni. Consideriamo ad esempio il seguente codice:

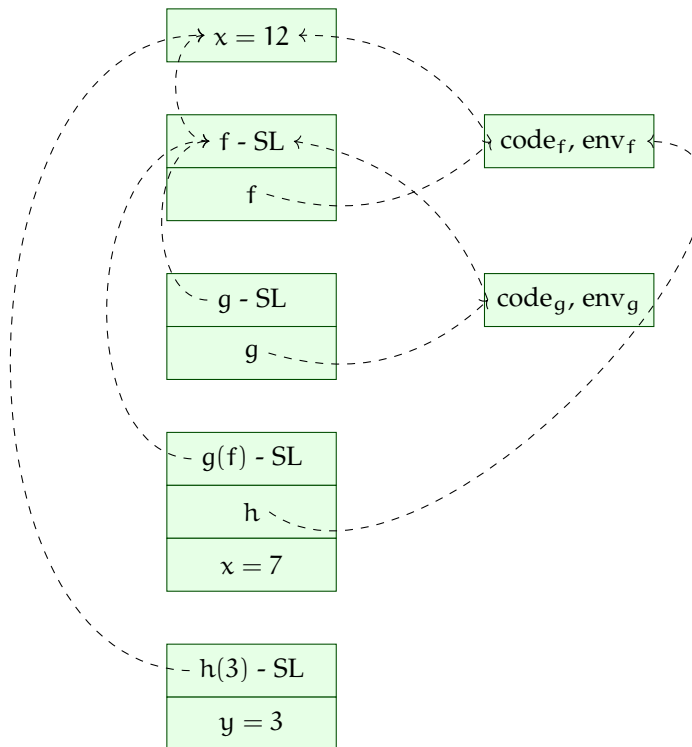


Figura 2: Struttura dei record di attivazione con le chiusure

```

let x = 4;
let f(y) = x*y;
let g(h) = let x = 7 in
            h(3) + x;
g(f);

```

La funzione f prende un parametro e lo moltiplica per x ; la funzione g prende una funzione h come parametro, dichiara $x = 7$ e restituisce il valore della funzione h valutata nel punto 3 più il valore di x .

La domanda che possiamo porci è: quale definizione di x deve essere usata nella chiamata $g(f)$?

Assumendo regole di scoping statico la chiamata $h(3)$ deve usare la definizione di f data poco sopra, quindi il valore della x deve essere 4, mentre quando sommiamo x al risultato di $h(3)$ la x in questione è quella legata dal **let**, quindi deve essere uguale a 7.

Quindi nel chiamare la funzione f dobbiamo tener traccia sia del suo codice (cioè ciò che la funzione fa) sia dell'ambiente in cui è stata dichiarata (memorizzato dal puntatore di catena statica), in modo da poter risolvere eventuali riferimenti non locali.

Il valore di una funzione è quindi una coppia

$\langle \text{codice funzione, ambiente di dichiarazione} \rangle$.

Una tale coppia si chiama **chiusura** (o in inglese **closure**).

Quando una funzione viene invocata si alloca sullo stack il suo record di attivazione, si va a considerare la chiusura relativa alla funzione e si usa come *puntatore di catena statica* il puntatore all'ambiente di dichiarazione, come possiamo vedere in Figura 2 per quanto riguarda il programma precedente.

CHIUSURE RICORSIVE Nel caso di funzioni ricorsive l'ambiente di dichiarazione non è l'ambiente puntato dallo static link, ma è la funzione stessa:

infatti se la funzione è ricorsiva il nome della funzione stessa deve essere nell'ambiente di dichiarazione.

CHIUSURE COME VALORI DI RITORNO L'ultimo caso è quello in cui vogliamo restituire una funzione: in questo caso il valore restituito è una chiusura $\langle env, code \rangle$, ma dobbiamo segnalare che l'ambiente di dichiarazione *env* non può essere distrutto fino a quando la funzione può essere usata. Questo meccanismo si chiama **retention**.

FUNZIONI IN AMBIENTE DINAMICO Se il linguaggio sfrutta lo scoping dinamico non è necessario usare i puntatori di catena statici (che non esistono), poiché le associazioni non locali vengono sempre risolte nel record di attivazione precedente in senso dinamico. In particolare non è neanche necessario usare le chiusure, poiché ci basta risalire la pila dei record di attivazione.

Implementazione dei record di attivazione

L'implementazione effettiva dei record di attivazione varia da linguaggio a linguaggio, ma ci sono delle linee guida generali.

Innanzitutto tutte le dichiarazioni vengono solitamente trasformate dal compilatore in una coppia $\langle \text{livello}, \text{offset} \rangle$:

- il *livello* indica il livello lessicale in cui è dichiarata la variabile, ovvero in quale blocco si trova;
- l'*offset* ci permette di distinguere la variabile dalle altre variabili dichiarate nello stesso blocco.

Le variabili nel codice vengono poi trasformate in un'altra coppia di valori: il secondo è l'*offset* dichiarato prima, mentre il primo rappresenta la differenza tra il livello in cui la variabile viene usata e il livello in cui la variabile viene dichiarata; ci dice quindi di quanto dobbiamo risalire la catena statica per trovare la variabile.

2.8 PASSAGGIO DEI PARAMETRI

Vogliamo ora studiare più in dettaglio le modalità di passaggio dei parametri nei vari linguaggi di programmazione.

Innanzitutto lo scopo dell'*astrazione parametrica* è quella di poter usare lo stesso frammento di codice con dati diversi a seconda della necessità: in particolare la *funzione* viene dichiarata con una lista di **parametri formali**, che rappresentano dei placeholder per i valori che verranno sostituiti in seguito. I valori passati alla funzione (chiamati **parametri attuali**) sono nella forma di una lista di espressioni: ognuno di essi verrà sostituito al posto del corrispondente parametro formale.

Il passaggio dei parametri è quindi una specie di *dichiarazione dinamica*: al momento della chiamata di funzione viene fatto un binding tra il parametro formale e il valore ottenuto dalla valutazione del parametro attuale. Esistono quindi diverse modalità di passaggio di parametri:

- passaggio per costante;
- passaggio per riferimento;
- passaggio di oggetti;
- passaggio di funzioni o procedura.

PASSAGGIO PER COSTANTE Nel caso di passaggio per costante il valore passato deve essere un valore *immutable*: il sottoprogramma non può modificarlo. Questo tipo di passaggio è il tipo di tutti i linguaggi funzionali, ma è presente anche in alcuni linguaggi imperativi.

PASSAGGIO PER RIFERIMENTO In questo caso il parametro formale ha come tipo un valore modificabile: viene passata una locazione di memoria, quindi possiamo direttamente modificare il suo contenuto e gli effetti si ripercuotono all'esterno della procedura (fenomeno dell'**aliasing**). Spesso il passaggio per riferimento non è realizzato concretamente, ma simulato attraverso passaggio del *valore* di puntatori (come nel caso del C).

PASSAGGIO DI OGGETTI In questo caso il parametro formale ha come tipo un *puntatore ad un oggetto*: non possiamo modificare il puntatore, ma attraverso esso possiamo modificare l'oggetto puntato.

PASSAGGIO DI FUNZIONI Il parametro formale ha come tipo una funzione, procedura o classe: l'espressione passata come parametro attuale deve essere il nome di una classe o anche un'espressione che ritorna una funzione, dunque la sua valutazione è una chiusura. Questo tipo particolare di parametro può essere solamente passato ulteriormente ad altre funzioni, oppure può essere attivato (la funzione può essere chiamata o la classe può essere istanziata tramite la **new**). Siccome nei linguaggi imperativi e object-oriented solitamente le funzioni non sono esprimibili, questo tipo di passaggio di parametri è presente principalmente nei linguaggi funzionali.

Esistono anche altre tecniche di passaggio di parametri, come

- **passaggio per valore o risultato**: in questi caso oltre all'ambiente si valuta anche la memoria;
- **passaggio per nome**: il parametro attuale non viene valutato.

PASSAGGIO PER VALORE Questo meccanismo coinvolge valori modificabili e pertanto non esiste nei linguaggi funzionali puri: consiste nel creare una variabile con il nome del parametro formale ed assegnare ad essa il valore del parametro attuale. In questo modo dobbiamo quindi coinvolgere la memoria, in quanto scriviamo nella locazione occupata dal parametro formale il valore del parametro attuale. In particolare, se l'assegnamento è implementato correttamente l'ambiente non viene coinvolto, non viene creato aliasing e non si possono avere effetti laterali (cioè eventuali modifiche al parametro formale non si ripercuotono all'esterno della funzione).

PASSAGGIO PER VALORE-RISULTATO Nel caso si voglia modificare anche il parametro attuale modificando il parametro formale, ma senza ricorrere agli effetti laterali diretti del passaggio per riferimento, si può implementare un meccanismo di passaggio per valore-risultato: come nel caso precedente quando la funzione viene chiamata si assegna il valore del parametro attuale alla variabile che rappresenta il parametro formale, ma alla fine della funzione viene anche fatto l'assegnamento inverso, per cui la variabile del parametro attuale viene modificata.

Esiste anche un passaggio per risultato, dove viene fatto solo il secondo assegnamento.

Osserviamo che il passaggio per valore-risultato è diverso dal passaggio per riferimento: infatti nel primo caso si crea una copia del valore del parametro attuale, per cui durante l'esecuzione della funzione i due parametri si riferiscono a due variabili (e quindi a due locazioni di memoria) diverse che possono evolvere indipendentemente; invece nel secondo caso le due variabili si riferiscono alla stessa locazione di memoria (*aliasing*) e quindi

ogni modifica fatta al parametro formale si ripercuote immediatamente sul parametro attuale.

PASSAGGIO PER NOME Nel caso del passaggio per nome l'espressione passata in corrispondenza del parametro attuale non viene valutata immediatamente: verrà valutata ogni volta che si incontra (eventualmente) un'occorrenza del parametro formale durante l'esecuzione della funzione. L'espressione non valutata diventa quindi una *chiusura* contenente l'espressione e l'ambiente al momento della definizione: ogni volta che sarà necessario valutarla, dovremo farlo nell'ambiente di definizione.

Un'espressione passata per nome è quindi equivalente ad una funzione senza parametri: ogni volta che vogliamo valutarla dobbiamo fare lo stesso procedimento che seguiamo nel caso di un'applicazione di funzione. Una peculiarità delle espressioni passate per nome è che vanno sempre valutate nell'ambiente di definizione, anche nel caso di linguaggi con scoping dinamico.

Questo meccanismo di passaggio è presente in linguaggi come l'ALGOL e il LISP, è alla base del meccanismo di valutazione lazy di Haskell ed è simulabile in altri linguaggi funzionali (come OCaml) tramite funzioni senza parametri.

2.9 LINGUAGGIO DIDATTICO

In questa sezione ci occuperemo di implementare in OCaml un piccolo nucleo di un linguaggio funzionale con scoping statico, contenente

- interi e booleani,
- poche operazioni di base,
- il costrutto if-then-else,
- il costrutto let-in,
- funzioni e funzioni ricorsive.

Definiamo innanzitutto la sintassi astratta e i tipi esprimibili a runtime dal linguaggio.

```
(* ide represents a variable identifier *)
type ide = string

(* expr is the Abstract Syntax Tree of the language *)
type expr =
  (* base values *)
  | LitTrue
  | LitFalse
  | LitInt of int
  (* variables *)
  | Den of ide
  (* basic operation *)
  | Add of expr * expr
  | Sub of expr * expr
  | Times of expr * expr
  | IsZero of expr
  | Equals of expr * expr
  (* if-then-else construct *)
  | IfThenElse of expr * expr * expr
  (* let-in construct *)
  | Let of ide * expr * expr
```

```

(* anonymous function *)
| Fun of ide * expr
(* function application *)
| Apply of expr * expr

(* Runtime values *)
type evT =
| Int of int
| Bool of bool
| Unbound

```

Ricordiamo inoltre che è necessario definire un ambiente di valutazione, altrimenti non possiamo memorizzare i valori delle variabili usate.

```

(* The polymorphic environment *)
type 'v env = (string * 'v) list

(* empty_env returns an empty environment *)
let empty_env : evT env = [("", Unbound)]
(* bind adds a new binding to a pre-existing environment *)
let bind (env : evT env) (x : ide) (v : evT) = (x, v)::env
(* lookup looks for the value of the variable identified by x in the
   ↪ environment *)
let rec lookup (env : evT env) (x : ide) =
  match env with
  | [] -> Unbound
  | (a, v)::_ when a = x -> v
  | _::e -> lookup e x

```

Implementiamo anche una semplicissima forma di typechecking dinamico.

```

let typecheck (t : string) (v : evT) =
  match t with
  | "int" ->
    ( match v with
    | Int(_) -> true
    | _ -> false )
  | "bool" ->
    ( match v with
    | Bool(_) -> true
    | _ -> false )
  | _ -> failwith "not a valid type"

```

Possiamo dunque iniziare a definire la funzione di valutazione: innanzitutto sappiamo che dovrà prendere in input un'espressione, un ambiente di valori e dovrà restituire un valore. Avrà quindi la seguente forma:

```

let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...

```

Iniziamo ad aggiungere mano mano le varie espressioni possibili all'interprete.

Valori letterali

Valutare letterali è ovvio.

```

let rec eval (e : expr) (env : evT env) : evT =
  match e with

```

```

| LitInt(n) -> Int n
| LitTrue  -> Bool true
| LitFalse -> Bool false

```

Variabili

Per valutare una variabile è sufficiente prendere dall'ambiente il suo valore.

```

let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Den id          -> lookup env id

```

Operazioni di base

Per valutare le operazioni di base usiamo delle funzioni ausiliarie.

```

let int_add (n : evT) (m : evT) : evT =
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a + b)
  | -, -, -, -                -> raise TypeError

let int_sub (n : evT) (m : evT) : evT =
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a - b)
  | -, -, -, -                -> raise TypeError

let int_times (n : evT) (m : evT) : evT =
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a * b)
  | -, -, -, -                -> raise TypeError

let is_zero (n : evT) : evT =
  match typecheck "int" n, n with
  | true, Int a -> Bool (a = 0)
  | -, -        -> raise TypeError

let int_equals (v1 : evT) (v2 : evT) : evT =
  match typecheck "int" v1, typecheck "int" v2, v1, v2 with
  | true, true, Int n, Int m -> Bool (n = m)
  | -, -, -, -                -> raise TypeError

let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Add (e1, e2) -> int_add (eval e1 env) (eval e2 env)
  | Sub (e1, e2) -> int_sub (eval e1 env) (eval e2 env)
  | Times (e1, e2) -> int_times (eval e1 env) (eval e2 env)
  | IsZero e -> is_zero (eval e env)
  | Equals (e1, e2) -> int_equals (eval e1 env) (eval e2 env)

```


Costrutto condizionale

Per valutare il costrutto condizionale ci rifacciamo alle regole di semantica date nelle sezioni precedenti: in particolare le regole operazionali per l'if-then-else sono

$$\frac{\text{env} \triangleright \text{cond} \Rightarrow \text{True} \quad \text{env} \triangleright e_1 \Rightarrow v_1}{\text{env} \triangleright \text{IfThenElse}(\text{cond}, e_1, e_2) \Rightarrow v_1}$$

$$\frac{\text{env} \triangleright \text{cond} \Rightarrow \text{False} \quad \text{env} \triangleright e_1 \Rightarrow v_1 \quad \text{env} \triangleright e_2 \Rightarrow v_2}{\text{env} \triangleright \text{IfThenElse}(\text{cond}, e_1, e_2) \Rightarrow v_2}$$

Possiamo quindi esprimere la regola in OCaml aggiungendo la seguente clausola:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | IfThenElse (cond, e1, e2) ->
    ( let evalCond = eval cond env in
      match typecheck "bool" evalCond, evalCond with
      | true, Bool true   -> eval e1 env
      | true, Bool false  -> eval e2 env
      | _                 -> raise TypeError )
```

Costrutto let

Anche per quanto riguarda il let ci rifacciamo alla regola operazionale:

$$\frac{\text{env} \triangleright e_1 \Rightarrow v' \quad \text{env}[x := v'] \triangleright e_2 \Rightarrow v}{\text{env} \triangleright \text{Let}(x, e_1, e_2) \Rightarrow v}$$

da cui la corrispondente regola dell'interprete è

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Let (id, e1, e2) ->
    let newEnv = (bind env id (eval e1 env)) in
    eval e2 newEnv
```

Astrazione funzionale

Vogliamo ora aggiungere al linguaggio un meccanismo per costruire funzioni anonime. Per semplicità consideriamo funzioni con un singolo parametro, dato che grazie al *currying* possiamo ottenere gratuitamente le funzioni con più parametri.

Come descritto nella sintassi astratta, il costruttore di una funzione anonima è **Fun of** *ide* * *expr*. I due parametri richiesti sono

- il nome del parametro della funzione;
- il corpo della funzione, che è un'altra espressione.

Come abbiamo visto precedentemente nel caso di linguaggi funzionali con scoping statico valutare una funzione equivale ad ottenere una *chiusura*. Aggiungiamo quindi il tipo delle chiusure ai tipi esprimibili:

```
type evT =
  | Int    of int
  | Bool   of bool
```

```
| Closure of ide * expr * evT env
| Unbound
```

Una chiusura è quindi data da

- il nome del parametro formale (di tipo `ide`);
- il codice della funzione dichiarata (di tipo `expr`);
- l'ambiente di dichiarazione della funzione (di tipo `evT env`).

La regola operativa è quindi ovvia:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Fun (id, body) -> Closure (id, body, env)
```

Applicazione di funzione

Avere la possibilità di dichiarare funzioni senza poterle valutare è inutile, quindi sfruttiamo il costrutto `Apply` per applicare le funzioni dichiarate con il costrutto `Fun`.

La regola per l'applicazione di funzione è la seguente:

$$\frac{\begin{array}{c} \text{env} \triangleright f \Rightarrow \text{Closure}(\text{id}, \text{body}, \text{funDeclEnv}) \\ \text{env} \triangleright \text{arg} \Rightarrow v_{\text{arg}} \\ \text{funDeclEnv}[\text{id} := v_{\text{arg}}] \triangleright \text{body} \Rightarrow v \end{array}}{\text{env} \triangleright \text{Apply}(f, \text{arg}) \Rightarrow v}$$

Analizziamo un momento cosa ci dice la regola: supponiamo di voler valutare in un ambiente `env` l'espressione `Apply(f, arg)`. Allora dobbiamo

- (1) valutare `f` e verificare che sia una chiusura, formata da
 - `id`: il nome del parametro formale,
 - `body`: il corpo della funzione,
 - `funDeclEnv`: l'ambiente di dichiarazione della funzione;
- (2) valutare il parametro attuale `arg`, ottenendo un valore `varg`;
- (3) valutare nell'ambiente di dichiarazione della funzione arricchito dal legame tra il parametro formale `id` e il valore del parametro attuale `varg` il corpo della funzione (ovvero `body`).

A questo punto è semplice tradurre la regola in codice OCaml:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Apply(f, arg) ->
    ( match eval f env with
      | Closure(id, body, funDeclEnv) ->
        let valArg = eval arg env in
        let newEnv = bind funDeclEnv id valArg in
        eval body newEnv
      | _ -> failwith "not a functional value" )
```

Funzioni ricorsive

L'interprete definito finora non ci consente di dichiarare o valutare funzioni ricorsive: infatti come abbiamo visto precedentemente nelle chiusure ricorsive deve essere presente anche la funzione stessa. Per poter dichiarare funzioni ricorsive dobbiamo quindi estendere la sintassi del nostro linguaggio con un nuovo costrutto:

```
type expr =
| ...
| LetRec of ide * ide * expr * expr
```

Una dichiarazione di funzione ricorsiva è quindi una versione particolare del costrutto `let` formata da

- il nome della funzione,
- il nome del parametro,
- il corpo della funzione,
- l'espressione in cui applichiamo la funzione ricorsiva.

Abbiamo anche bisogno di estendere i tipi esprimibili per includere le chiusure ricorsive:

```
type evT =
| ...
| RecClosure of ide * ide * expr * evT env
```

In questo caso i parametri sono sempre nome della funzione e del parametro, corpo della funzione e ambiente al momento della dichiarazione della funzione.

La regola operativa per valutare un `let rec` è equivalente a quella del `let`, soltanto che dobbiamo costruire la chiusura ricorsiva prima di valutare l'espressione principale.

$$\frac{\text{env}[f := r] \triangleright \text{letBody} \Rightarrow v \quad \text{dove } r = \text{RecClosure}(f, \text{id}, \text{funBody}, \text{env})}{\text{env} \triangleright \text{LetRec}(f, \text{id}, \text{funBody}, \text{letBody}) \Rightarrow v}$$

La corrispondente regola in OCaml diventa

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | LetRec(f, id, funBody, letBody) ->
    let r = RecClosure(f, id, funBody, env) in
    let newEnv = bind env f r in
    eval letBody newEnv
```

Anche in questo caso dobbiamo permettere l'applicazione di una funzione ricorsiva ad un parametro: dobbiamo quindi espandere il costrutto `Apply` con una nuova regola operativa.

$$\frac{\begin{array}{l} \text{env} \triangleright f \Rightarrow \text{closure} \\ \text{env} \triangleright \text{arg} \Rightarrow v_{\text{arg}} \\ \text{newEnv} \triangleright \text{body} \Rightarrow v \\ \text{dove } \text{closure} = \text{RecClosure}(\text{funId}, \text{id}, \text{body}, \text{funDeclEnv}) \\ \text{newEnv} = \text{funDeclEnv}[f := \text{closure}, \text{id} := v_{\text{arg}}] \end{array}}{\text{env} \triangleright \text{Apply}(f, \text{arg}) \Rightarrow v}$$

La regola ci dice che per valutare un'espressione della forma **Apply**(f, arg) dobbiamo

- valutare f e usare questa regola se il risultato è una chiusura ricorsiva;
- valutare il parametro formale arg, ottenendo il valore v_{arg} ;
- calcolare l'ambiente in cui valuteremo il corpo della funzione, che è l'ambiente di dichiarazione (funDeclEnv) insieme al legame tra la funzione f e la sua chiusura e a quello tra il nome del parametro formale (id) e il valore del parametro attuale (v_{arg});
- valutare il corpo della funzione in questo nuovo ambiente.

Possiamo quindi finalmente ampliare l'interprete:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Apply(f, arg) ->
    ( match eval f env with
      | Closure(id, body, funDeclEnv) -> ...
      | RecClosure(funId, id, body, funDeclEnv) as funClosure ->
        let actualVal = eval arg env in
        (* bind between f and the closure *)
        let recEnv = bind funDeclEnv funName funClosure in
        (* bind between the formal parameter and the actual value *)
        let actualEnv = bind recEnv param actualVal in
          eval funBody actualEnv
      | _ -> failwith "not a functional value" )
```

3

SEMANTICA DEI
LINGUAGGI
OBJECT-ORIENTED

3.1 CLASSI E METODI

Vogliamo quindi estendere lo studio dei blocchi e delle astrazioni procedurali in generale allo studio di classi e oggetti. La differenza principale con quanto studiato prima è che le classi ci forniscono un meccanismo per creare un ambiente e una memoria *permanenti*, cioè che sopravvivono alla creazione dell'oggetto, e *accessibili* e *utilizzabili* da chiunque possieda il loro meccanismo di accesso.

Sappiamo che un tale meccanismo di astrazione può essere utilizzato nella maggior parte dei linguaggi orientati agli oggetti attraverso la parola chiave **new**: essa viene usata per *attivare* un nuovo oggetto e con esso il suo ambiente locale.

Implementazione delle variabili di istanza

Nell'ambiente locale di un oggetto sono certamente contenute le sue variabili di istanza. La strategia che utilizziamo per implementare le variabili di istanza è un semplice ambiente locale statico contenente i legami tra i nomi delle variabili (e il loro descrittore di tipo) e i loro valori. Ad esempio se considerassimo il codice

```
class A {  
    int a1;  
    int a2;  
}  
A obj = new A(4, 5);
```

l'ambiente locale statico di `obj` è formato da due campi (uno per ogni variabile di istanza) che contengono i due binding, più dello spazio per il descrittore di tipo della classe `A`.

Tuttavia sappiamo che il meccanismo dell'ereditarietà ci consente di avere variabili non dichiarate esplicitamente in una classe, ma ereditate dalla classe padre, come nel seguente esempio:

```
class A {  
    int a1;  
    int a2;  
}  
class B extends A {  
    int b;  
}
```

Ogni oggetto di tipo `B` conterrà all'interno del suo ambiente locale (e in particolare all'inizio) i campi relativi alle variabili ereditate.

In questo modo risolviamo molto semplicemente alcuni possibili problemi:

- la gestione dell'ereditarietà singola è immediata;
- la gestione dello *shadowing* (la sottoclasse ha variabili con lo stesso nome della classe padre) è altrettanto immediata;

- se sono previsti meccanismi di controllo statico è facile implementare un accesso diretto ai vari campi dell'oggetto tramite un *indirizzo di base* e un *offset*.

Metodi e dispatching

I metodi di una classe sono comuni a tutti gli oggetti di una data classe, quindi viene naturale memorizzarli in un'unica area di memoria a cui si può accedere conoscendo il tipo della variabile che stiamo considerando.

Tuttavia, l'implementazione dei metodi non può essere così immediata. Infatti consideriamo il caso delle *interfacce* in Java:

```
List<String> l = ...
l.add("ciao");
```

Essendo `List` solo un'interfaccia, essa non implementa direttamente il metodo `add`: esso viene implementato dalle varie classi che rispettano l'interfaccia `List` in modi diversi, quindi per eseguirlo dobbiamo conoscere la classe dell'oggetto `l`. L'invocazione di un metodo deve quindi necessariamente *passare per l'oggetto*: questa nozione viene chiamata **dynamic dispatch**.

La soluzione al problema delle interfacce è quindi quello di associare alla classe una tabella (detta **tabella dei metodi** oppure **dispatch table/vector**) che contiene il binding dei metodi e il descrittore di tipo della classe: ogni oggetto della classe contiene un riferimento alla tabella dei metodi, dunque possiamo eseguire ogni metodo della classe anche quando lavoriamo con le interfacce.

I metodi in sé sono implementati come normali funzioni, quindi quando sono invocati creano un record di attivazione sullo stack con i campi già visti in precedenza. Tuttavia ogni metodo deve poter accedere alle variabili di istanza dell'oggetto: l'oggetto diventa quindi un *parametro implicito* del metodo, e il riferimento ad esso è dato dal puntatore `this`.

Come risolviamo i problemi di ereditarietà per quanto riguarda i metodi? Esistono due soluzioni.

SOLUZIONE SMALLTALK Nel linguaggio Smalltalk i dispatch vector di classi in gerarchia formano una *lista di tabelle*: la tabella dei metodi di una classe contiene solo i metodi effettivamente implementati dalla classe; per accedere ai metodi della classe padre bisogna risalire la lista di tabelle fino a trovare il metodo cercato. Questa soluzione è particolarmente semplice, ma porta ad un discreto overhead in run-time.

SHARING STRUTTURALE La soluzione di **sharing strutturale** è quella adottata da linguaggi come il C++ e il Java: ogni classe ha la sua tabella dei metodi contenente i nomi dei metodi e dei *puntatori* al codice. In questo modo una classe che eredita un metodo deve semplicemente inizializzare il puntatore al codice già creato precedentemente. In particolare ogni metodo è identificabile staticamente attraverso il suo offset nella tabella dei metodi, quindi l'overhead a runtime della prima soluzione è azzerato.

Metodi statici

I metodi statici vengono implementati come se fossero delle procedure dei linguaggi imperativi: non potendo accedere alle variabili di istanza non hanno bisogno del puntatore `this` all'oggetto, e quindi sono semplicemente delle funzioni nell'ambiente globale.

Compilazione separata

Alcuni linguaggi (come il Java) implementano il meccanismo della *compilazione separata delle classi*: la compilazione di una classe produce del codice che viene caricato dalla macchina astratta del linguaggio *dinamicamente* quando il programma effettua un riferimento alla classe (**class loading**).

In questo caso gli offset non possono essere calcolati staticamente in quanto si possono fare modifiche non visibili alla struttura delle classi.

3.2 JAVA VIRTUAL MACHINE

Studiamo ora i meccanismi di funzionamento del Java ed in particolare la Java Virtual Machine (JVM).

Un'applicazione Java viene eseguita da un programma chiamato **Java Runtime**, contenente la JVM e la Java Class Library (JCL), che contiene tutte le librerie del Java.

La Java Virtual Machine è la macchina astratta del Java: essa descrive il *class file format*, ovvero il formato e i vincoli (strutturali e sintattici) che devono essere rispettati da tutti i file `.class` a prescindere dalla macchina su cui vengono compilati.

I file prodotti dal compilatore `.javac` non sono quindi specifici per un solo tipo di macchina, ma sono universali: è la JVM che si occupa di compilarli nel linguaggio **bytecode**, che è il linguaggio macchina usato dalla JVM. In particolare quindi la JVM è formata da

- il *loader*, usato per caricare i file;
- il *verifier*, usato per controllare che i file `.class` caricati rispettino le specifiche e non siano quindi malevoli/malformati;
- in *linker*, che collega le librerie e i vari file tra loro, restituendo il bytecode;
- il *bytecode interpreter*, che esegue il programma.

Perché il doppio livello di compilazione?

- Innanzitutto perché la specifica della JVM non prescrive come deve essere implementata, quindi in questo modo possono essere implementate forme diverse della macchina virtuale.
- Inoltre le macchine hardware non possono avere a che fare con alberi di sintassi astratta, ma operano direttamente sui registri: astraendo sulla macchina hardware possiamo generare un linguaggio vicino alla macchina che però opera su strutture superiori all'architettura.

In particolare il bytecode Java sfrutta uno stack invece dei registri: le operazioni sono fatte sullo stack e lo modificano in modo incrementale, nel senso che ogni operazione viene eseguita sugli elementi che si trovano in testa allo stack e il risultato viene messo in testa allo stack. Lo stack degli operandi viene utilizzato per

- trasmettere i parametri ai metodi;
- restituire il risultato di un metodo;
- memorizzare i risultati intermedi delle operazioni;
- memorizzare le variabili locali.

La JVM può quindi essere descritta come una macchina astratta, stack-based e multi-threaded. In particolare ogni thread della JVM contiene

- un **program counter**, contenente l'indirizzo dell'istruzione corrente del metodo in esecuzione;
- uno stack chiamato **ambiente**, contenente i record di attivazione (chiamati **frame**) dei metodi. Ogni frame contiene
 - un array con le variabili locali (**local variable array**);
 - uno spazio per il valore di ritorno;
 - lo stack degli operandi;
 - un riferimento alle informazioni di runtime, contenute nella **constant pool**.

In particolare il local variable array contiene tutte le variabili locali del metodo e tutti i parametri, incluso **this** nel caso di metodi non statici. Nel caso di metodi non statici quindi la posizione 0 è riservata al puntatore **this**, mentre nel caso di metodi statici le variabili locali e i parametri partono dall'indice 0.

Ogni frame inoltre contiene un puntatore ad una struttura particolare, denominata **constant pool**: essa contiene il descrittore di tipo associato alla classe dove è definito il metodo in esecuzione, e viene usato per fare il linking dinamico. Infatti quando una classe Java viene compilata, tutti i riferimenti a variabili e metodi nella classe sono memorizzati nella constant pool come riferimenti simbolici (nel senso che non indicano effettivamente una locazione fisica di memoria, ma solo logica).

I riferimenti simbolici così introdotti possono essere risolti con le seguenti strategie:

- possono essere risolti quando le classi vengono caricate, con una strategia *eager*;
- possono essere risolti la prima volta che durante l'esecuzione si incontra il riferimento, con una strategia chiamata *lazy resolution*; in particolare se il riferimento è per una classe che ancora non è stata caricata, il programma si occupa di caricarla con una strategia lazy (**lazy class loading**);
- ogni riferimento risolto diventa un offset rispetto alla struttura di memorizzazione a runtime.

Lo spazio dedicato alle classi nella JVM viene chiamato **Class Area**. Per ogni classe essa contiene:

- riferimento al classloader;
- la constant pool;
- i dati relativi alle variabili di istanza;
- i dati relativi ai metodi;
- il codice dei metodi.

Il bytecode generato dal compilatore javac viene memorizzato in un file `.class` contenente il bytecode dei metodi della classe e la constant pool della classe: quest'ultima viene usata nel class loading per risolvere i riferimenti simbolici nominati precedentemente.

Gli offset di accesso ai metodi non possono invece essere determinati staticamente: essi vengono determinati dinamicamente la prima volta che si trova un riferimento all'oggetto, mentre per eventuali accessi successivi si utilizza l'offset già calcolato.

3.3 GARBAGE COLLECTION

Abbiamo visto studiando la JVM che durante l'esecuzione di un programma Java si usano tre tipi di memoria:

- una *static area* dove vengono allocati contenuti definiti a tempo di compilazione;
- un *run-time stack*, che è di dimensione variabile e viene usato per allocare i vari record di attivazione dei sottoprogrammi e dei blocchi;
- uno *heap*, di dimensione fissa o variabile, usato per allocare oggetti e strutture dati dinamiche tramite il costrutto `new`.

Nella prima area vi sono generalmente le variabili globali, le variabili locali dei sottoprogrammi (non ricorsivi), le costanti determinate a tempo di compilazione, le tabelle usate dal supporto a runtime per effettuare ad esempio il type checking e in generale tutte le entità che hanno un indirizzo di memoria assoluto che viene mantenuto per tutta l'esecuzione del programma.

Nello stack, come abbiamo visto diverse volte, abbiamo un record di attivazione per ogni istanza di un sottoprogramma con le informazioni relative a quella determinata istanza.

Lo **heap** invece è la regione di memoria in cui possiamo allocare e deallocare blocchi di memoria in momenti arbitrari. Esso è necessario quando il linguaggio permette

- allocazione esplicita di memoria a run-time (come con il costrutto `new` o con la primitiva `malloc` del C);
- oggetti di dimensioni variabili;
- oggetti con vita non esprimibile da una struttura dati LIFO come una pila.

Tuttavia una corretta gestione dello heap non è banale: vediamo innanzitutto delle possibili implementazioni.

IMPLEMENTAZIONE CON BLOCCHI DI DIMENSIONE FISSA La prima implementazione dello heap può essere la seguente: scegliamo un'area di memoria da usare come heap e suddividiamola in blocchi di dimensione fissa (abbastanza limitata). Questi blocchi vengono usati per formare una lista (detta *lista libera*) contenente tutti i blocchi che non sono stati allocati dal programma.

Inizialmente la lista contiene tutti i blocchi dello heap, ma ogni volta che viene allocato un oggetto tramite la `new` alcuni di questi blocchi vengono rimossi dalla lista libera e usati per contenere le informazioni dell'oggetto creato. Quando la memoria occupata dall'oggetto viene liberata i corrispondenti blocchi vengono restituiti alla lista libera e possono essere riutati per successive allocazioni.

Questo metodo è semplice, ma porta ad una grande frammentazione dello heap poiché la dimensione fissa dei blocchi non è necessariamente la più adatta in ogni circostanza.

HEAP CON BLOCCHI VARIABILI In questa seconda implementazione inizialmente lo heap è formato da un unico blocco. Ad ogni richiesta di allocazione si cerca il blocco di dimensione opportuna usando una delle seguenti due strategie:

- **first fit**: si sceglie il primo blocco grande abbastanza per contenere l'oggetto da allocare;

- **best fit:** tra i blocchi grandi a sufficienza per contenere l'oggetto da allocare si sceglie il blocco più piccolo (con il minor spreco di spazio quindi).

Se il blocco scelto è molto più grande del necessario viene diviso in due e la parte inutilizzata viene restituita alla lista libera; stessa cosa succede quando il contenuto di un blocco viene deallocato, ed in particolare se un blocco adiacente è libero i due vengono fusi in un unico blocco.

Garbage collection

In alcuni linguaggi (come ad esempio il C/C++) la gestione della memoria viene lasciata interamente al programmatore: ogni blocco di memoria allocato dinamicamente va deallocato dinamicamente. Seppure questa scelta garantisca la massima flessibilità ed efficienza, un uso scorretto della memoria fa emergere diversi problemi, come ad esempio

- **memory leaks**, che avvengono quando strutture vengono allocate e mai deallocate;
- **dangling reference**, quando dei puntatori continuano a puntare ad aree di memoria deallocate;
- **heap fragmentation**, che si ha quando la lista libera dello heap è molto frammentata a seguito di diverse allocazioni e deallocazioni.

In più, da un punto di vista più concettuale, la gestione esplicita della memoria viola il principio di astrazione: è necessario quindi che questo processo sia automatizzato.

Il **Garbage Collector** è un componente della macchina virtuale che esegue un programma di un dato linguaggio: ad esempio linguaggi come Lisp, Scheme, Java, Haskell contengono un garbage collector nella loro macchina virtuale, mentre le macchine virtuali del C e C++ generalmente no.

Vediamo i principali problemi da risolvere.

FRAMMENTAZIONE Esistono due tipi di frammentazione: la **frammentazione interna** e la **frammentazione esterna**.

Si parla di **frammentazione interna** quando viene allocato un blocco di dimensione maggiore di quella strettamente necessaria: una parte dello spazio allocato è sprecato.

Si parla invece di **frammentazione esterna** quando vogliamo allocare un blocco di una certa dimensione x e la quantità di memoria libera ce lo consente, ma la memoria libera è frammentata e ognuno dei blocchi liberi non è abbastanza grande per allocare un blocco di dimensione x .

GESTIONE DELL'ALLOCAZIONE Dobbiamo inoltre decidere quale tipo di strategia usare per allocare i blocchi: la strategia first fit è più veloce ma spreca più memoria, mentre la strategia best fit è più lenta ma più efficiente nella gestione della memoria.

IDENTIFICAZIONE DEI BLOCCHI DA DEALLOCARE Un'altra scelta che dobbiamo fare è come decidere quali blocchi vanno deallocati. Nel caso di una deallocazione esplicita è semplice, in quanto basta seguire il puntatore passato come parametro alla funzione `free` (nel caso del C): tuttavia questo meccanismo può causare i problemi visti precedentemente e quindi si preferisce avere una gestione automatica.

Nel caso di deallocazione automatica diciamo che una porzione di memoria è recuperabile (nel senso che può essere deallocata) se non è più rag-

giungibile dal resto del programma. Dobbiamo quindi capire come definire precisamente il concetto di raggiungibilità.

Tecniche di garbage collection

Studiamo ora le principali tecniche di garbage collection.