Programmazione II

Luca De Paulis

21 novembre 2020

INDICE

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 <exp> $\Longrightarrow$ 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`:

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

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:

$$\frac{\text{Eval}(\texttt{f}) = \texttt{fun x\_1 ... x\_n = e} \quad (\forall i : \text{Eval}(\texttt{e\_i}) = v_i) \quad \text{subst}(\texttt{e}, \texttt{x\_1}, \ldots, \texttt{x\_n}, v_1, \ldots, v_n) = \texttt{e}' \quad \text{Eval}(\texttt{e}') = v}{\text{Eval}(\texttt{f e\_1 ... e\_n}) = v}.$$

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

- `f` viene valutata ad una funzione (`fun`) che prende n parametri (`x_1 ... x_n`) e restituisce un'espressione che dipende da questi parametri (`e`);
- i parametri attuali `e_1 ... e_n` vengono valutati a dei valori $v_1, \ldots, v_n$;
- 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:
$$\text{Eval}(\texttt{plus}) = \texttt{fun x y = x + y}.$$
- 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 l'`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 $J_1, \ldots, J_n$, J delle sentenze. Una **regola di inferenza** è un'implicazione della forma

$$\frac{J_1 \quad \ldots \quad J_n}{J},$$

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

I judgment $J_1, \ldots, J_n$ 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:

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

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 : 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:

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

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` signfica 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:

$$\frac{}{\Gamma \vdash \texttt{Const : ok}} \qquad \frac{\texttt{\textbackslash Gamma \textbackslash proves E1 : ok} \quad \texttt{\textbackslash Gamma \textbackslash proves E2 : ok}}{\texttt{\textbackslash Gamma \textbackslash proves (Times E1 E2) : ok}} \qquad \frac{\texttt{\textbackslash Gamma \textbackslash proves E1 : ok}}{\texttt{\textbackslash Gamma \textbackslash proves (P}}$$

$$\frac{\texttt{x} \in \Gamma}{\Gamma \vdash \texttt{x : ok}} \qquad \frac{\Gamma \vdash \texttt{E1 : ok} \quad \Gamma \cup \{\texttt{x}\} \vdash \texttt{E2 : ok}}{\Gamma \vdash \texttt{(Let x E1 E2)}}$$

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

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

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

$$\mathrm{I} \coloneqq \{\texttt{e} : \varnothing \vdash \texttt{e} : \texttt{ok}\};$$

- F come l'insieme dei valori interi:

$$\mathrm{F} \coloneqq \{\texttt{Int n} : \texttt{n} \in \mathbb{N}\}.$$

Iniziamo a studiare le regole di valutazione.

TIMES Facciamo l'esempio della regola del costrutto `Times`:

$$\frac{}{\texttt{(Times (Int n) (Int m))} \to \texttt{Int (n * m)}}$$

$$\frac{e1 \to e1'}{\texttt{(Times e1 e2)} \to \texttt{(Times e1' e2)}}$$

$$\frac{e2 \to e2'}{\texttt{(Times (Int n) e2)} \to \texttt{(Times (Int n) e2')}}$$

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 `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`:

$$\frac{}{\texttt{(Let x (Int n) e2)} \to \texttt{e2[x} \coloneqq \texttt{(Int n)]}}$$

$$\frac{e1 \to e1'}{\texttt{(Let x e1 e2)} \to \texttt{(Let x e1' e2)}}$$

Questa regola ci dice che

- se abbiamo valutato completamente `e1` al valore `(Int n)` allora lo step consiste nel restituire l'espressione `e2` dove tutte le occorrenze di `x` vengono sostituite da occorrenze di `(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 $E = \{\texttt{e} : \Gamma \vdash \texttt{e} : \texttt{ok}\}$,
- V è l'insieme dei valori interi,
- alcune regole di valutazione sono le seguenti:

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

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

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

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:

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

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

$$\text{env} : \texttt{Ide} \to \texttt{Values} \cup \texttt{Unbound}$$

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 env `[x := v]`, che indica la funzione che si comporta esattamente come env per ogni $y \in \texttt{Ide} \setminus \{\texttt{x}\}$, mentre $\text{env}(\texttt{x}) = \texttt{v}$.

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

$$S = \{\, \rho \triangleright \texttt{e} \;:\; \rho \in \texttt{Env}, \texttt{e} \in \texttt{Exp}, \varnothing \vdash \texttt{e} \;:\; \texttt{ok} \,\}.$$

  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:

$$\frac{\text{env}(\texttt{id}) = v \in V}{\text{env} \triangleright \texttt{(Den id)} \Rightarrow v}$$

$$\frac{\text{env} \triangleright \texttt{e1} \Rightarrow v_1 \quad \text{env} \triangleright \texttt{e2} \Rightarrow v_2 \quad v_1 \cdot v_2 = v}{\text{env} \triangleright \texttt{(Times e1 e2)} \Rightarrow v}$$

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

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 TYPE SYSTEM

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.