



AA 2014-2015

18. Semantica operativa per type-checking ed eliminazione di ricorsione

1



A cosa servono i tipi?

- 👁️ A garantire che un programma abbia comportamenti corretti rispetto ai valori sui quali opera
 - non si somma un valore di tipo intero con un valore di tipo stringa: $25 + \text{"abcd"}$
 - le variabili sono dichiarate prima del loro utilizzo all'interno di un programma
- 👁️ A ottimizzare la generazione del codice
 - viene occupata esattamente la memoria che serve per i valori

2

Controlli statici



- La compilazione in un codice intermedio più efficiente è solo uno degli aspetti significativi dell'utilizzo di tecniche di compilazione
- Oltre agli aspetti di analisi sintattica e ottimizzazione del codice, il compilatore effettua diversi controlli di analisi statica sulla struttura del codice
- Esempio: **Type checking** per l'analisi statica dell'uso corretto dei tipi associati ai costrutti linguistici
- Vediamo un semplice esempio basato sul nostro linguaggio di espressioni

3

Type checking



- Cosa controlla un type checker (ovviamente dipende dal tipo di linguaggio)
 - chiamata di una funzione con un numero errato di parametri attuali
 - uso di variabili non dichiarate
 - funzioni che non restituiscono valori
 - "array out of bound indices" (pensate al C)
 - :

4

Come descrivere un type checker?



- Associazione di tipo ai costrutti di un linguaggio di programmazione è un'asserzione J della forma

$$J = env \triangleright e \Rightarrow T$$

- Nell'ambiente env l'espressione e ha tipo T
- L'ambiente contiene le associazioni tra identificatori e tipi

5

Regole di tipo



- Le regole di tipo costituiscono un *proof system*

$$\frac{J_1 \quad \dots \quad J_k}{J}$$

- Le regole sono definite per induzione strutturale sulla sintassi del linguaggio
 - esattamente come abbiamo operato nel caso dell'interprete

6

Regole di tipo



- Una regola del tipo

$$\frac{J_1 \quad \dots \quad J_k}{J}$$

può essere letta dal basso verso l'alto

- per controllare (check) J dobbiamo controllare $J_1 \dots J_k$

7

Type checking vs. Type inference



- Type checking:** data una asserzione di tipo $env \Rightarrow e : T$, si deve determinare se l'asserzione può essere derivata applicando le regole di tipo
- Type inference:** dato un programma e , si deve determinare un tipo T e un opportuno ambiente env tale che $env \Rightarrow e : T$ può essere derivato

8

Type checking vs. Type inference



- ✎ Java, C#: principalmente type checking
 - anche se alcuni aspetti sui generici possono richiedere type inference
- ✎ ML, Haskell: type inference

9

Typed expression



- ✎ Modifichiamo leggermente la sintassi del nostro linguaggio introducendo costanti con tipo intero e booleano e un costrutto condizionale

```
type tyexpr =  
  | CstI of int  
  | CstB of bool  
  | Var of string  
  | Let of string * tyexpr * tyexpr  
  | Prim of string * tyexpr * tyexpr  
  | If of tyexpr * tyexpr * tyexpr
```

10

Typed expression



☞ Con questa sintassi possiamo costruire programmi che non sono tipati correttamente

- la guardia del condizionale non è booleana
- un operatore è applicato a argomenti di tipo errato
- i due rami del condizionale non hanno lo stesso tipo

```
Let("z", CstI 17,  
    Let("y", CstB true,  
        If(Var "z",  
            Prim("+", Var "y", Var "y"),  
            Var "z")))
```

Non è un valore booleano !

Somma di booleani !

I tipi



☞ Il tipo è una proprietà che caratterizza i valori che sono manipolati dal programma

- “la variabile X contiene valori di tipo stringa”
- “int è il tipo del valore restituito dalla funzione F”

☞ L’informazione di tipo è uno strumento utile nello sviluppo di programmi perché permette di evitare alcuni errori di programmazione

Tipi: statico vs. dinamico



- ✎ Il **controllo statico dei tipi** permette di verificare la correttezza dal programma rispetto alle proprietà dei tipi staticamente, cioè prima di mandare il programma in esecuzione
 - Ocaml, Java, C#, ... sono linguaggi con controllo statico
- ✎ Il **controllo dinamico dei tipi** verifica le proprietà dei tipi a tempo di esecuzione
 - Lisp, Smalltalk, JavaScript, ... sono linguaggi con controllo dinamico

13

Java: controllo dei tipi dinamico



- ✎ La divisione statico/dinamico non è sempre netta
- ✎ Consideriamo Java
 - le operazioni di “downcast” sono verificate a run-time e possono sollevare eccezioni.
 - l’accesso agli elementi degli array in Java sono controllati a run-time
- ✎ La **ClassCastException** viene sollevata dal controllo dinamico dei tipi in Java: codice “unsafe” rispetto all’uso dei tipi non è dunque mai eseguito in Java

14

Esempio: checking dinamico in Java



```
class Calculation {
    public int f(int x) { return x; }
}
class Person {
    public String getName( ) { return "persona"; }
}
class Main {
    public static void main(String[] args) {
        Object o = new Calculation( );
        System.out.println(((Person) o).getName( ));
    }
}
```

**Exception in thread "main" java.lang.ClassCastException:
Calculation at Main.main(example.java:12)**

15

Meglio tardi che mai...



- ☞ D'altra parte, in C o C++ il tipo base degli array (int, float) è controllato staticamente, ma
 - l'accesso agli elementi degli array non viene controllato neanche a runtime
- ☞ Anche il downcasting viene controllato a runtime
 - provate a riscrivere in C++ il programma Java che abbiamo visto in precedenza, e vedrete che il programma viene eseguito con esiti non prevedibili
- ☞ Il punto: C++ non effettua controlli dinamici di tipo

16

Typed expression



- ✎ Per controllare l'utilizzo dei tipi staticamente abbiamo bisogno di una struttura di implementazione che associa ai valori presenti nel programma il loro tipo
- ✎ *Ambiente di tipo*: è la struttura che associa alle variabili presenti nel programma il loro tipo
 - è un ambiente dove i valori sono i tipi
 - nel contesto dei compilatori viene anche chiamata **Tabella dei Simboli**

17

I tipi del nostro esempio OCaml



- ✎ Obiettivo: definire una funzione che data una espressione con tipi (`typexpr`)
 - restituisce il tipo dell'espressione, se è ben tipata
 - altrimenti lancia un'eccezione

```
type typ =  
  | TypI           (* int *)  
  | TypB           (* bool *)
```

18

Serve un ambiente dei tipi



- ... e binding (string, typ)
- Definizione identica a quella degli ambienti già visti, che contenevano binding (string, int)
- Sfruttiamo i tipi polimorfi di OCaml

```
type 't env = (string * 't) list

let rec lookup amb y = match amb with
| (il,e1) :: amb1 ->
  if y = il then e1 else lookup amb1 y
| [] -> failwith("wrong typed env")
```

19

Type checking (I)



```
let rec typ (e : tyexpr) (env : typ env) : typ =
  match e with
  | CstI i -> TypI
  | CstB b -> TypB
  | Var x -> lookup env x
  | Prim(ope, e1, e2) ->
    let t1 = typ e1 env in
    let t2 = typ e2 env in
    (match (ope, t1, t2) with
    | ("*", TypI, TypI) -> TypI
    | ("+", TypI, TypI) -> TypI
    | ("-", TypI, TypI) -> TypI
    | ("=", TypI, TypI) -> TypB
    |("<", TypI, TypI) -> TypB
    |("&", TypB, TypB) -> TypB
    | _ -> failwith "unknown op, or type error")
```

20

Type checking (II)



```
| Let(x, eRhs, letBody) ->
  let xTyp = typ eRhs env in
  let letBodyEnv = (x, xTyp) :: env in
  typ letBody letBodyEnv
| If(e1, e2, e3) ->
  (match typ e1 env with
  | TypB -> let t2 = typ e2 env in
             let t3 = typ e3 env in
             if t2 = t3 then t2
             else failwith "If: branch types differ"
  | _    -> failwith "If: condition not boolean")
```

21

Type checking



- ☞ Si noti come il type checker utilizzi le medesime regole della semantica operativa, ma utilizzi come dominio dei valori l'insieme dei tipi e come ambiente l'ambiente dei tipi
- ☞ Type checker: interprete del linguaggio su un dominio astratto di valori

22

Semantica operativa del Let



Regola di esecuzione

$$\frac{env \triangleright ehrs \Rightarrow xval \quad env[xval / x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let } x = ehrs \text{ in } ebody \Rightarrow v}$$

Regola di type checking

$$\frac{tenv \triangleright ehrs \Rightarrow tval \quad tenv[tval / x] \triangleright ebody \Rightarrow t}{tenv \triangleright \text{Let } x = ehrs \text{ in } ebody \Rightarrow t}$$

23

Implementazione in Ocaml del Let



```
typ Let(x, eRhs, letBody) tenv ->  
  let xTyp = typ eRhs tenv in  
  let letBodyEnv = (x, xTyp) :: tenv in  
  typ letBody letBodyEnv
```

```
eval Let(x, erhs, ebody) env ->  
  let xval = eval erhs env in  
  let env1 = (x, xval) :: env in  
  eval ebody env1
```

24

Esercizio



- Mettete assieme tutte le cose che abbiamo visto per creare un **sistema di valutazione di espressioni tipate** che
 - effettua il type checking, lanciando un'eccezione se l'espressione non è ben tipata
 - compila l'espressione in un codice intermedio senza variabili
 - esegue il codice intermedio ottenuto testando il codice prodotto su alcuni programmi
- Utilizzate liberamente il codice OCaml presentato a lezione (messo in linea sulla pagina del corso)

25

Cosa abbiamo ottenuto?



- La semantica operativa (**eval**) è l'interprete del linguaggio
 - definito in modo ricorsivo
 - ✓ `eval Prim("-", e1, e2) env -> eval e1 env - eval e2 env`
 - utilizzando la ricorsione di OCaml (ling. di implementazione)
- La semantica operativa sul dominio dei tipi (**typ**) definisce il type checker del linguaggio
- È una vera implementazione? Certamente, ma...
 - **eliminando la ricorsione** dall'interprete ne otteniamo una versione più a basso livello, più vicina a una "vera" implementazione

26

Ricorsione



- ✎ La ricorsione può essere rimpiazzata con l'iterazione ma sono necessari degli stack per simulare il passo ricorsivo
 - a meno di definizioni ricorsive con una struttura molto semplice (tail recursion)
- ✎ NB. la struttura ricorsiva di **eval** ripropone quella del dominio sintattico delle espressioni (composizionalità)
- ✎ Il dominio delle espressioni non è tail recursive
 - Prim of string * exp * exp | ...
- ✎ Morale: per eliminare la ricorsione serve una rappresentazione esplicita degli stack

27

Macchine virtuali



- ✎ Ci sono fondamentalmente due modi principali per implementare una macchina virtuale
 - Stack VM (JVM, C# CLR, OCaml)
 - Register VM (C, LUA)
- ✎ La differenza tra i due approcci consiste nel meccanismo utilizzato per il trattamento del trasferimento dati (recupero e memorizzazione di operandi e risultati)

28

Stack machine



- Una macchina a stack implementa i registri con uno stack. Gli operandi dell'unità logica aritmetica (ALU) sono sempre i primi due registri dello stack e il risultato della ALU viene memorizzato nel registro in cima alla pila
- Il set di istruzioni utilizza la Notazione Polacca Inversa (Reverse Polish Notation) e le operazioni fanno riferimento solamente alla pila e non ai registri o alla memoria principale

29

Esempi di Stack VM



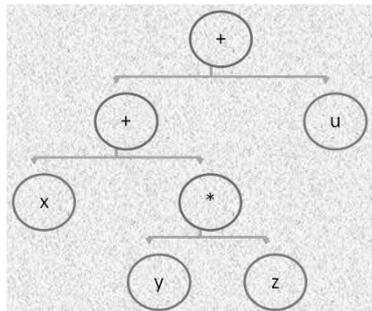
- UCSD Pascal p-machine (sostanzialmente la Burroughs stack machine del 1961)
- Java virtual machine
- VES (Virtual Execution System) del Common Intermediate Language (CIL) di Microsoft .NET
- Forth virtual machine
- Adobe Machine per PostScript
- Sun SwapDrop (linguaggio di progr. per Smartcard)
- Per i curiosi
 - http://en.wikipedia.org/wiki/Stack_machine

30

Esempio di valutazione



$x + y * z + u$



Notazione Polacca Inversa

$x y z * + u +$

codice compilato per
una Stack Machine
basata su NPI

```
push x
push y
push z
multiply
add
push u
add
```

31

Valutazione ricorsiva di espressioni



Consideriamo il linguaggio di espressioni intere

```
type expr =
  | CstI of int
  | Var of string
  | Prim of string * expr * expr
```

e l'interprete OCaml già visto

```
let rec eval e (env : (string * int) list) : int =
  match e with
  | CstI i          -> i
  | Var x          -> lookup env x
  | Prim("+", e1, e2) -> eval e1 env + eval e2 env
  | Prim("*", e1, e2) -> eval e1 env * eval e2 env
  | Prim("-", e1, e2) -> eval e1 env - eval e2 env
  | Prim _        -> failwith "unknown primitive";
```

32

Eliminare la ricorsione



- La funzione **eval** ha due argomenti (espressione da valutare e ambiente) e calcola il risultato (un intero) ricorsivamente
- Obiettivo: definire una nuova funzione di valutazione per espressioni nella sintassi vista (non in NPI), **che non sia ricorsiva**, quindi utilizzando degli stack
- NB. l'ambiente non è modificato nelle chiamate ricorsive
- L'informazione da memorizzare in stack è costituita da
 - le sotto-espressioni da valutare
 - il valore calcolato per le sotto-espressioni

33

Eliminare la ricorsione: idea



- Usiamo una pila di espressioni “colorate” (il **continuation stack contSt**)
 - **espressione rossa**: deve essere valutata
 - **espressione blu**: è cominciata la valutazione delle sotto-espressioni, e se ne attendono i risultati per applicare l'operatore
- Una pila di di valori interi (**tempSt**)
 - contiene i risultati temporanei
- NB. sono le informazioni che inserivamo nello stack della Abstract Stack Machine di Java
- Vediamo l'algoritmo su un esempio

34

La valutazione di una espressione



$+(*(X,2),-(Y,3))$ nell'ambiente $\{X \rightarrow 5, Y \rightarrow 7\}$

$+(*(X,2),-(Y,3))$ []	$+(*(X,2),-(Y,3))$ $*(X,2)$ $-(Y,3)$ []	$+(*(X,2),-(Y,3))$ $*(X,2)$ $-(Y,3)$ Y 3 []	$+(*(X,2),-(Y,3))$ $*(X,2)$ $-(Y,3)$ Y []
$+(*(X,2),-(Y,3))$ $*(X,2)$ $-(Y,3)$ 3 7 []	$+(*(X,2),-(Y,3))$ $*(X,2)$ 4 []	$+(*(X,2),-(Y,3))$ $*(X,2)$ X 2 4 []	$+(*(X,2),-(Y,3))$ $*(X,2)$ X 4 2 []
$+(*(X,2),-(Y,3))$ $*(X,2)$ 4 2 5 []	$+(*(X,2),-(Y,3))$ 4 10 []	14 []	

35

Definizioni ausiliarie: gli stack



Usiamo un semplice tipo polimorfo per gli stack: in una situazione reale avremmo usato un TdA, definito usando i moduli OCaml

```

type 'a stack = Empty | Push of 'a stack * 'a
let emptystack = Empty
let isempty p = (p = emptystack)
let push p a = Push(p,a)
let pop p = match p with
| Push(p1, _) -> p1
| Empty -> failwith "pop on empty stack"
let top p = match p with
| Push(_, a) -> a
| Empty -> failwith "top on empty stack"

```

36

Espressioni colorate, ambienti e stack ausiliari



Elementi di `coloredExpr` sono espressioni intere colorate

```
type coloredExpr =  
  | Blue of expr  
  | Red of expr
```

Ambiente polimorfo con lookup

```
type 't env = (string * 't) list  
  
let rec lookup ide (amb: 't env) = match amb with  
  | (ide1, vall) :: amb1 ->  
    if ide = ide1 then vall else (lookup ide amb1)  
  | [] -> failwith (" ^ide^" not bound);;  
  
(* Stack di coloredExpr, continuazioni *)  
let contSt = ref Empty      (* NB. variabili in OCaml! *)  
  
(* Stack di interi, risultati temporanei *)  
let tempSt = ref Empty
```

37

L'interprete iterativo (1)



```
let evalIt ((e:expr), (rho: int env)) =  
  contSt := push !contSt (Red(e));  
  while not(isempty(!contSt)) do  
    match top(!contSt) with  
    | Red(x) ->  
      (contSt := pop !contSt;  
       match x with  
       | CstI a -> tempSt := push !tempSt a;  
       | Var str ->  
         tempSt := push !tempSt (lookup str rho);  
       | Prim (op, e1, e2) ->  
         (contSt := push !contSt (Blue(x));  
          contSt := push !contSt (Red(e1));  
          contSt := push !contSt (Red(e2)))  
      ) (* continua *)
```

38

L'interprete iterativo (2)



```
| Blue(x) ->
  (contSt := pop !contSt;
   match x with
   | Prim (op, _, _) ->
     let e1 = top !tempSt in
     tempSt := pop !tempSt;
     let e2 = top !tempSt in
     (tempSt := pop !tempSt;
      match op with
      | "+" -> tempSt := push !tempSt (e1 + e2)
      | "-" -> tempSt := push !tempSt (e1 - e2)
      | "*" -> tempSt := push !tempSt (e1 * e2)
      | str -> failwith (" unknown op "^str)
      )
     )
   | _ -> failwith (" cerroneous blue expr ")
   )
done;
let res = top !tempSt in tempSt := pop !tempSt; res ;;
```

39

Testando l'interprete iterativo



```
(* Esempio di valutazione: l'espressione dell'animazione *)
(* +( *(X,2),-(Y,3)) nell'ambiente {X -> 5, Y -> 7} *)
(* L'espressione *)
let myExpr = Prim("+", Prim("*", Var("X"), CstI(2)),
                 Prim("-", Var("Y"), CstI(3)));;

(* L'ambiente *)
let myEnv = [("X",5);("Y",7)];;

(* La valutazione: ci aspettiamo 14 *)
evalIt(myExpr, myEnv);;

(* Test per vedere errori *)
(* Variable non bound *)
evalIt(Prim("*", Var("Z"), CstI(2)), [("X",5);("Y",7)])
(* Operatore non conosciuto *)
evalIt(Prim("/", Var("X"), CstI(2)), [("X",5);("Y",7)])
```

40

Stack Machine



- ✎ Abbiamo realizzato una macchina virtuale basata sulla nozione di stack: la struttura di memoria in cui sono memorizzati gli operandi è una pila
- ✎ Le operazioni sono effettuate in tre passi
 - estrarre dati dalla pila
 - elaborare i dati
 - inserire i risultati sulla pila
- ✎ Sia JVM che MS-CLI sono stack machine (più articolate di quella che abbiamo visto...)

41

Parentesi



- ✎ Quello che abbiamo visto è la compilazione di un semplice linguaggio in una macchina virtuale simile alla SECD machine di Peter Landin e alla Functional Abstract Machine di Luca Cardelli per la realizzazione di ML
 - lettura interessante
 - http://en.wikipedia.org/wiki/SECD_abstract_machine
- ✎ Il progetto e la realizzazione dei linguaggi di programmazione fanno emergere i fondamenti teorici dell'informatica

42