SEMANTICA OPERAZIONALE PER TYPE CHECKING E ELIMINAZIONE DI RICORSIONE

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

Typed expressions

Modifichiamo leggermente la sintassi del nostro linguaggio introducento 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
```

3

Typed Expressions

Con questa sintassi possiamo costruire programmi che non sono tipati correttamente, perché

- la guardia del condizionale non è booleana, oppure
- un operatore è applicato a argomenti di tipo errato, oppure
- 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

5

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 dei tipi
- Il **controllo dinamico dei tipi** verifica le proprietà dei tipi a tempo di esecuzione
  - Lisp, Smalltalk JavaScript sono linguaggi con controllo dinamico dei tipi

6

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. Pertanto codice "unsafe" rispetto all'uso dei tipi non viene mai eseguito in Java.

7

Esempio: type 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)

8

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 esguito con esiti non prevedibili.
- Il punto: C++ non effettua controlli dinamici di tipo

9

Typed Expressions

- 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***

10

Definiamo i tipi del nostro semplice esempio di espressioni OCaml

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

Obiettivo: definire una funzione che data una espressione con tipi (`typexpr`)

- Restituisce il tipo dell'espressione, se è ben tipata
- Altrimenti lancia un'eccezione

11

Abbiamo bisongo di un ambiente dei tipi, con bindings (string, typ)

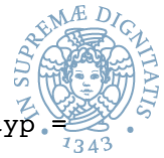
Definizione identica a quella degli ambienti già visti, che contenevano bindings (string, int). Sfruttiamo i tipi polimorfi di Ocaml:

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

let rec lookup amb y = match amb with
     | (i1,e1) :: amb1 ->
            if y = i1 then e1 else lookup amb1 y
     | [] -> failwith("wrong typed env")
```

12

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")
```


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")
```

Type Checking

- Si noti che il type checker utilizza essenzialmente le medesime regole della semantica operazionale, ma come dominio dei valori utilizza l'insieme dei tipi e come ambiente utilizza l'ambiente dei tipi

15

Regole del Let: semantica operazionale

- Regola di esecuzione:

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

- Regola di type checking:

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

16

Regole del Let: implementazione in OCaml


```
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
```

17

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
  - Testare il codice prodotto su alcuni programmi
- Utilizzare liberamente il codice OCaml presentato a lezione (messo in linea sulla pagina del corso)

18

Cosa abbiamo ottenuto?

- La semantica operazionale (**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 (linguaggio di implementazione)
- La semantica operazionale sul dominio dei tipi (**typ**) definisce il typechecker del linguaggio
- E' una vera implementazione? Certamente, ma…
  - **eliminando la ricorsione** dall'interprete ne otteniamo una versione più a basso livello più vicina ad una "vera" implementazione

19

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)
- Nota: 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

20

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)

21

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.

22

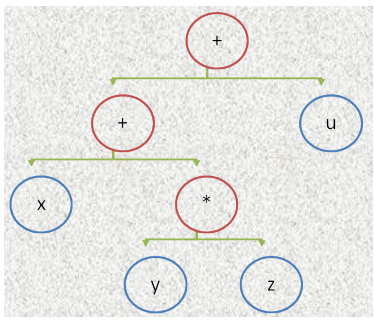
Esempi di Stack VM

- 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 programmazione per Smartcard)
- Per I curiosi:
  - **http://en.wikipedia.org/wiki/Stack_machine**

23

Esempio di valutazione

x + y * z + u



Notazione Polacca Inversa:
x y z * + u +

Codice Compilato per Stack Machine basato su NPI

push x
push y
push z
multiply
add
push u
add

24

Valutazione ricorsiva di espressioni

Consideriamo il seguente linguaggio di semplici 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";
```

25

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
- Si noti che l'ambiente non viene modificato nelle chiamate ricorsive
- L'informazione da memorizzare in opportuni stack è costituita da
  - Le sotto-espressioni da valutare
  - il valore calcolato per le sotto-espressioni

26

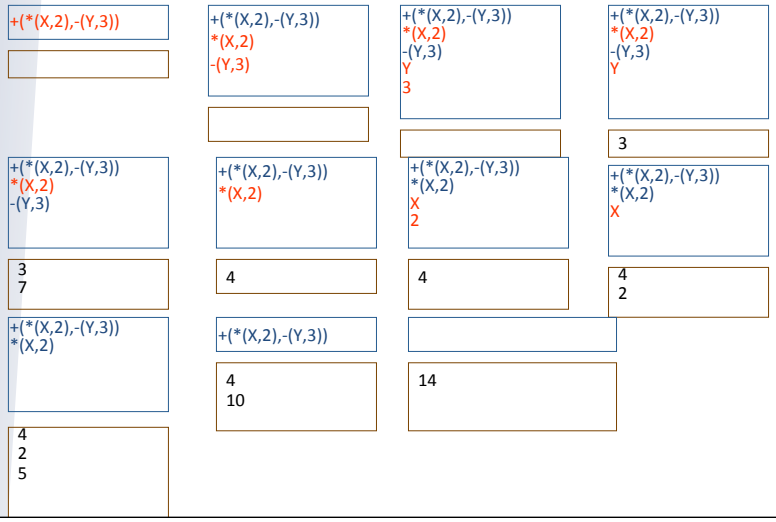
Eliminare la ricorsione: idea

- Usiamo una pila di espressioni "colorate" (il **continuation stack `contSt`**)
  - **Espressione rossa**: deve essere valutata
  - **Espressione blu**: ho cominciato la valutazione delle sotto-espressioni, e aspetto i risultati per applicare l'operatore
- Una pila di di valori interi (**`tempSt`**)
  - Contiene i risultati temporanei
- Nota: sono le informazioni che inserivamo nello stack della Abstract Stack Machine di Java
- Vediamo l'algoritmo su un esempio

27

La valutazione di una espressione

`+(*(X,2),-(Y,3))` nell'ambiente **`{X -> 5, Y -> 7}`**



28

Definizione ausiliarie: gli stack

Usiamo un semplice tipo polimorfo per gli stack. In 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 a p = Push(a,p)
   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"
```


Defausiliarie: 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, val1) :: amb1 ->
      if ide = ide1 then val1 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
```

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 *)
```


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 (" erroneous blue expr")
           )
    done;
   let res = top !tempSt in tempSt := pop !tempSt; res ;;
```

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)])
```


Stack Machine

- Abbiano realizzato una macchina virtuale basata sulla nozione di stack: la struttura di memoria in cui sono memorizzati gli operandi è una pila
- 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…)

Parentesi

- Quello che abbiamo visto è la compilazione di un semplice linguaggio in una macchina virtuale simile alla SECD machine di Peter Landin e la Functional Abstract Machine di Luca Cardelli per la realizzazione di ML
  - Lettura intressante:

  http://en.wikipedia.org/wiki/SECD_abstract_machine
- Il progetto e la realizzazione dei linguaggi di programmazione fanno emergere i fondamenti teorici dell'informatica

35