

Controllo di sequenza: espressioni e comandi

1

Di cosa parleremo

- espressioni pure (senza blocchi e funzioni)
 - regola di valutazione, operazioni strette e non strette
- un frammento di linguaggio funzionale
 - semantica operativa
 - interprete iterativo
- comandi puri (senza blocchi e sottoprogrammi)
 - semantica dell'assegnamento
- un frammento di linguaggio imperativo
 - semantica operativa
 - interprete iterativo

2

Espressioni in sintassi astratta

- alberi etichettati
 - nodi
 - applicazioni di funzioni (operazioni primitive)
 - i cui operandi sono i sottoalberi
 - foglie
 - costanti o variabili (riferimenti a dati)
- il più semplice meccanismo per comporre operazioni
 - preso direttamente dalla matematica
- solo *espressioni pure*, che non contengono
 - definizioni di funzione (λ -astrazione)
 - applicazioni di funzione
 - introduzione di nuovi nomi (blocco)
- l'unico problema semantico interessante che riguarda la valutazione delle espressioni pure è quello della *regola di valutazione*

3

Le operazioni come funzioni

- le operazioni primitive sono in generale *funzioni parziali*
 - indefinite per alcuni valori degli input
 - errori "hardware"
 - overflow, divisione per zero
 - errori rilevati dal supporto a run time
 - errori di tipo a run time, accessi errati ad array, accessi a variabili non inizializzate, esaurimento memoria libera
 - nei linguaggi moderni tutti questi casi provocano il sollevamento di una eccezione
 - che può essere catturata ed eventualmente gestita
- alcune operazioni primitive sono *funzioni non strette*
 - una funzione è non stretta sul suo i -esimo operando, se ha un valore definito quando viene applicata ad una n -upla di valori, di cui l' i -esimo è indefinito

4

Espressioni: regole di valutazione

- regola interna
 - prima di applicare l'operatore, si valutano tutti i sottoalberi (sottoespressioni)
- regola esterna
 - è l'operatore che richiede la valutazione dei sottoalberi, se necessario
- le due regole di valutazione possono dare semantiche diverse
 - se qualcuna delle sottoespressioni ha valore "indefinito"
 - errore, non terminazione, sollevamento di una eccezione, ...
 - e l'operatore è non stretto
 - può calcolare un valore senza aver bisogno del valore di tutti gli operandi
 - quindi, può essere definito anche se qualcuno degli operandi è indefinito
- esempi di tipiche operazioni primitive non strette
 - Condizionale
 - If true then C1 else C2
 - or, and
 - True or E
- è molto utile avere la possibilità di definire funzioni (astrazioni procedurali) non strette
 - sarà un problema risolto con tecniche opportune di passaggio dei parametri (passaggio per nome)

5

Una operazione non stretta: il condizionale

```
if x = 0 then y else y/x
```

- in sintassi astratta
- ```
ifthenelse(=(x,0), y, /(y,x))
```
- usando la regola interna, valuto tutti e tre gli operandi
    - se  $x$  vale 0, la valutazione del terzo operando dà origine ad un errore
    - l'intera espressione ha valore indefinito
  - usando la regola esterna, valuto solo il primo operando
    - se  $x$  vale 0, valuto il secondo operando
    - il terzo operando non viene valutato e l'intera espressione ha un valore definito

6

## Una operazione non stretta: l'or

`true or "expr1"`

- in sintassi astratta

`or(true, "expr1")`

- usando la regola interna, valuto tutti e due gli operandi
  - se la valutazione del secondo operando dà origine ad un errore, l'intera espressione ha valore indefinito
  - in ogni caso, la valutazione di `"expr1"` è inutile!
- usando la regola esterna, valuto il primo operando
  - se questo vale `true`, non devo fare altro, ed il risultato è `true` qualunque sia il valore (anche indefinito) di `"expr1"`
  - altrimenti viene valutato `"expr1"`

7

## Regola esterna vs. regola interna

- la regola esterna
  - è sempre corretta
  - è più complessa da implementare, perché ogni operazione deve avere la propria "politica"
  - è necessaria in pochi casi, per le operazioni primitive
    - sono poche le operazioni primitive non strette
- la regola interna
  - non è in generale corretta per le operazioni non strette
  - è banale da implementare
- la soluzione più ragionevole
  - regola interna per la maggior parte delle operazioni
  - regola esterna per le poche primitive non strette

8

## Frammento funzionale: sintassi

```
type ide = string
type exp =
 | Eint of int
 | Ebool of bool
 | Den of ide
 | Prod of exp * exp
 | Sum of exp * exp
 | Diff of exp * exp
 | Eq of exp * exp
 | Minus of exp
 | Iszero of exp
 | Or of exp * exp
 | And of exp * exp
 | Not of exp
 | Ifthenelse of exp * exp * exp
```

9

## Domini semantici (denotazionale)

```
type eval =
 | Int of int
 | Bool of bool
 | Unbound

• l'implementazione funzionale dell'ambiente

module Funenv:ENV =
 struct
 type 't env = string -> 't
 let emptyenv(x) = function y -> x
 let applyenv(x,y) = x y
 let bind(r: 'a env) , (l:string), (e:'a) =
 function lu -> if lu = l then e else applyenv(r,lu)
 ...
 end
```

10

## Type checking

```
let typecheck (x, y) = match x with
 | "int" -> (match y with
 | Int(u) -> true
 | _ -> false)
 | "bool" -> (match y with
 | Bool(u) -> true
 | _ -> false)
 | _ -> failwith ("not a valid
type")
```

11

## La semantica operativa

```
let rec sem ((e:exp), (r:eval env)) =
 match e with
 | Eint(n) -> Int(n)
 | Ebool(b) -> Bool(b)
 | Den(i) -> applyenv(r,i)
 | Iszero(a) -> iszero(sem(a, r))
 | Eq(a,b) -> equ(sem(a, r), sem(b, r))
 | Prod(a,b) -> mult(sem(a, r), sem(b, r))
 | Sum(a,b) -> plus(sem(a, r), sem(b, r))
 | Diff(a,b) -> diff(sem(a, r), sem(b, r))
 | Minus(a) -> minus(sem(a, r))
 | And(a,b) -> et(sem(a, r), sem(b, r))
 | Or(a,b) -> vel(sem(a, r), sem(b, r))
 | Not(a) -> non((sem a r))
 | Ifthenelse(a,b,c) -> let g = sem(a, r) in
 if typecheck("bool",g) then
 (if g = Bool(true) then sem(b, r) else sem(c, r))
 else failwith ("nonboolean guard")
```

12 val sem : exp \* eval Funenv.env -> eval = <fun>

Assumendo note le operazioni primitive

## La semantica: commenti

...

```
| And(a,b) -> et(sem(a, r), sem(b, r))
| Or(a,b) -> vel(sem(a, r), sem(b, r))
```

- And e OR interpretati come funzioni strette

...

```
| Ifthenelse(a,b,c) -> let g = sem(a, r) in
 if typecheck("bool",g) then
 (if g = Bool(true) then sem(b, r) else sem(c, r))
 else failwith ("nonboolean guard")
```

- condizionale interpretato (ovviamente!) come funzione non stretta

13

## Operazioni primitive: parte 1

```
let minus x = if typecheck("int",x) then (match x with Int(y) -> Int(-y))
 else failwith ("type error")
```

```
let iszero x = if typecheck("int",x) then (match x with Int(y) -> Bool(y=0))
 else failwith ("type error")
```

```
let equ (x,y) = if typecheck("int",x) & typecheck("int",y)
 then (match (x,y) with (Int(u), Int(w)) -> Bool(u = w))
 else failwith ("type error")
```

```
let plus (x,y) = if typecheck("int",x) & typecheck("int",y)
 then (match (x,y) with (Int(u), Int(w)) -> Int(u+w))
 else failwith ("type error")
```

14

## Operazioni primitive: parte 2

```
let diff (x,y) = if typecheck("int",x) & typecheck("int",y)
 then (match (x,y) with (Int(u), Int(w)) -> Int(u-w))
 else failwith ("type error")

let mult (x,y) = if typecheck("int",x) & typecheck("int",y)
 then (match (x,y) with (Int(u), Int(w)) -> Int(u*w))
 else failwith ("type error")

let et (x,y) = if typecheck("bool",x) & typecheck("bool",y)
 then (match (x,y) with (Bool(u), Bool(w)) -> Bool(u & w))
 else failwith ("type error")

let vel (x,y) = if typecheck("bool",x) & typecheck("bool",y)
 then (match (x,y) with (Bool(u), Bool(w)) -> Bool(u or w))
 else failwith ("type error")

let non x = if typecheck("bool",x)
 then (match x with Bool(y) -> Bool(not y))
 else failwith ("type error")
```

15

## Cosa abbiamo ottenuto?

La semantica operativa e' un interprete

```
let rec sem ((e:exp), (r:eval env)) =
 match e with

 | Prod(a,b) -> mult(sem(a, r), sem(b, r))

 | Ifthenelse(a,b,c) -> let g = sem(a, r) in
 if typecheck("bool",g) then
 (if g = Bool(true) then sem(b, r) else sem(c, r))
 else failwith ("nonboolean guard")
val sem : exp * eval Funenv.env -> eval = <fun>
```

- definito in modo ricorsivo
  - utilizzando la ricorsione del metalinguaggio (linguaggio di implementazione)
- eliminando la ricorsione dall'interprete
  - ne otteniamo una versione più a basso livello
  - più vicina ad una "vera" implementazione

16



## Eliminare la ricorsione

```
let rec sem ((e:exp), (r:eval env)) =
 match e with

 | Prod(a,b) -> mult(sem(a, r), sem(b, r))

 | Ifthenelse(a,b,c) -> let g = sem(a, r) in
 if typecheck("bool",g) then
 (if g = Bool(true) then sem(b, r) else sem(c, r))
 else failwith ("nonboolean guard")
```

- la ricorsione può essere rimpiazzata con l'iterazione
    - in generale sono necessarie delle pile
    - a meno di definizioni ricorsive con una struttura molto semplice (tail recursion)
  - la struttura ricorsiva di sem ripropone quella del dominio sintattico delle espressioni (composizionalità)
    - il dominio delle espressioni non è tail recursive
- ```
type exp = .... | Prod of exp * exp | ...
```
- per eliminare la ricorsione servono delle pile

17

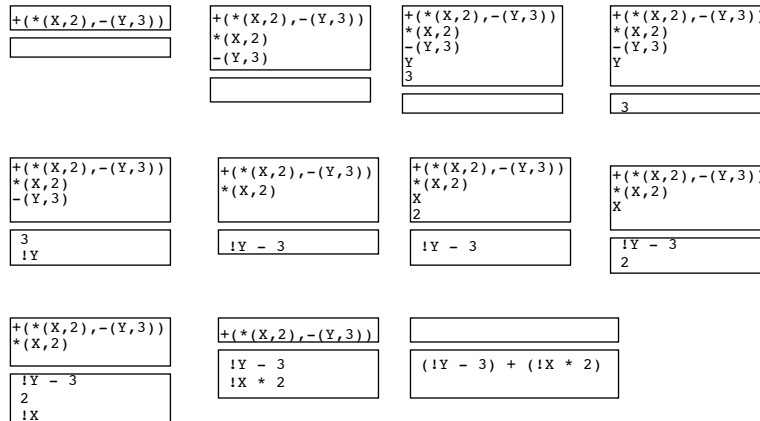
Come eliminiamo la ricorsione

- la funzione ricorsiva sem ha due argomenti
 - l'espressione
 - l'ambiente
- e calcola un risultato
 - un eval
- l'ambiente non viene mai modificato nelle chiamate ricorsive
- l'informazione da memorizzare in opportune pile per simulare la ricorsione è dunque
 - la (sotto)-espressione
 - il valore (eval) calcolato per la sotto-espressione
- una pila di espressioni etichettate
 - ad ogni istante, contiene l'informazione su "quello che deve ancora essere valutato"
 - **continuation**
- una pila di eval
 - ad ogni istante, contiene i risultati temporanei
 - **tempstack**
- vediamo l'algoritmo su un esempio
 - colori come etichette
 - sintassi "normale"

18

La valutazione di una espressione

$+(*(X,2),-(Y,3))$



19

Le strutture dell'interprete iterativo

```
let cframesize = 20
let tframesize = 20
```

```
type labeledconstruct =
  | Expr1 of exp
  | Expr2 of exp
```

```
let (continuation: labeledconstruct stack) =
  emptystack(cframesize, Expr1(Eint(0)))
```

```
let (tempstack: eval stack) =
  emptystack(tframesize, Unbound)
```

- una pila di espressioni etichettate
 - ad ogni istante, contiene l'informazione su "quello che deve ancora essere valutato"
 - **continuation**
- una pila di eval
 - ad ogni istante, contiene i risultati temporanei
 - **tempstack**

20

L'interprete iterativo

```
let sem ((e:exp), (rho:eval env)) =
  push(Expr1(e), continuation);
  while not(empty(continuation)) do
    (match top(continuation) with
    | Expr1(x) ->
      (pop(continuation); push(Expr2(x), continuation);
      (match x with
      | Iszero(a) -> push(Expr1(a), continuation)
      | Eq(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Prod(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Sum(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Diff(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Minus(a) -> push(Expr1(a), continuation)
      | And(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Or(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
      | Not(a) -> push(Expr1(a), continuation)
      | Ifthenelse(a,b,c) -> push(Expr1(a), continuation)
      | _ -> ()))
```

21

L'interprete iterativo

```
|Expr2(x) ->
  (pop(continuation); (match x with
  | Eint(n) -> push(Int(n), tempstack)
  | Ebool(b) -> push(Bool(b), tempstack)
  | Den(i) -> push(applyenv(rho,i), tempstack)
  | Iszero(a) -> let arg=top(tempstack) in pop(tempstack); push(iszero(arg), tempstack)
  | Eq(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(equ(firstarg, sndarg), tempstack)
  | Prod(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(mult(firstarg, sndarg), tempstack)
  | Sum(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(plus(firstarg, sndarg), tempstack)
  | Diff(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(diff(firstarg, sndarg), tempstack)
  | Minus(a) -> let arg=top(tempstack) in pop(tempstack); push(minus(arg), tempstack)
  | And(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(et(firstarg, sndarg), tempstack)
  | Or(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
    let sndarg=top(tempstack) in pop(tempstack); push(vel(firstarg, sndarg), tempstack)
  | Not(a) -> let arg=top(tempstack) in pop(tempstack); push(non(arg), tempstack)
  | Ifthenelse(a,b,c) -> let arg=top(tempstack) in pop(tempstack);
    if typecheck("bool", arg) then
      (if arg = Bool(true) then push(Expr1(b), continuation)
      else push(Expr1(c), continuation)
      else failwith ("type error"))))
  done;
  let valore= top(tempstack) in pop(tempstack); valore;;
val sem : exp * eval Funenv.env -> eval = <fun>
```

22

Cosa abbiamo imparato

- Una tecnica generale usata nei back-end dei compilatori nella fase di generazione del codice per ottenere un codice maggiormente efficiente.

23

Effetti laterali, comandi ed espressioni pure

- assumiamo che continuino ad esistere le espressioni
 - diverse dai comandi perché la loro semantica
 - non modifica lo store (non produce effetti laterali)
 - restituisce un valore (eval)
- tale approccio non è quello di C
 - in cui quasi ogni costrutto può restituire un valore e modificare lo stato
- la distinzione (semantica) tra espressioni e comandi è difficile da mantenere se si permette che comandi possano occorrere all'interno di espressioni (Java, ML), soprattutto in presenza di “operazioni definite dal programmatore” (funzioni)
- nel linguaggio didattico, forzeremo questa distinzione
 - permettendo “effetti laterali” solo in alcuni costrutti
 - che avranno una semantica diversa
- tali costrutti non esistono nel “frammento puro”

24

Un frammento di linguaggio imperativo: domini sintattici

```
type ide = string
type exp =
  | Eint of int
  | Ebool of bool
  | Den of ide
  | Prod of exp * exp
  | Sum of exp * exp
  | Diff of exp * exp
  | Eq of exp * exp
  | Minus of exp
  | Iszero of exp
  | Or of exp * exp
  | And of exp * exp
  | Not of exp
  | Ifthenelse of exp * exp * exp
  | Val of exp

type com =
  | Assign of exp * exp
  | Cifthenelse of exp * com list * com list
  | While of exp * com list
```

25

Domini semantici

- serve, oltre all'ambiente, la memoria
- ai domini semantici dei valori si aggiungono le locazioni
 - che decidiamo non essere nè esprimibili nè memorizzabili
- tre domini distinti: eval, dval, mval
 - con operazioni di "conversione"
 - esiste una funzione di valutazione semantica (semden) che calcola un dval invece che un eval

26

Il dominio store

- simile all'ambiente (polimorfo)

```
module type STORE =
sig
  type 't store
  type loc
  val emptystore : 't -> 't store
  val allocate : 't store * 't -> loc * 't store
  val update : 't store * loc * 't -> 't store
  val applystore : 't store * loc -> 't
end
module Funstore:STORE =
struct
  type loc = int
  type 't store = loc -> 't
  let (newloc,initloc) = let count = ref(-1) in
    (fun () -> count := !count +1; !count),
    (fun () -> count := -1)
  let emptystore(x) = initloc(); function y -> x
  let applystore(x,y) = x y
  let allocate(r: 'a store) , (e:'a) = let l = newloc() in
    (l, function lu -> if lu = l then e else applystore(r,lu))
  let update(r: 'a store) , (l:loc), (e:'a) =
    function lu -> if lu = l then e else applystore(r,lu)
end
```

27

I domini dei valori

```
exception Nonstorable
exception Nonexpressible
type eval = | Int of int
           | Bool of bool
           | Novalue
type dval = | Dint of int
           | Dbool of bool
           | Unbound
           | Dloc of loc
type mval = | Mint of int
           | Mbool of bool
           | Undefined
let evaltomval e = match e with
  | Int n -> Mint n
  | Bool n -> Mbool n
  | _ -> raise Nonstorable
let mvaltoeval m = match m with
  | Mint n -> Int n
  | Mbool n -> Bool n
  | _ -> Novalue
let evaltodval e = match e with
  | Int n -> Dint n
  | Bool n -> Dbool n
  | Novalue -> Unbound
let dvaltoeval e = match e with
  | Dint n -> Int n
  | Dbool n -> Bool n
  | Dloc n -> raise Nonexpressible
  | Unbound -> Novalue
```

28

Semantica operativa: espressioni

```

let rec sem ((e:exp), (r:dval env), (s: mval store)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> dvaltoeval(applyenv(r,i))
  | Iszero(a) -> iszero(sem(a, r, s))
  | Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))
  | Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))
  | Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))
  | Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))
  | Minus(a) -> minus(sem(a, r, s))
  | And(a,b) -> et (sem(a, r, s), sem(b, r, s))
  | Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))
  | Not(a) -> non(sem(a, r, s))
  | Ifthenelse(a,b,c) ->
    let g = sem(a, r, s) in
    if typecheck("bool",g) then
      (if g = Bool(true)
       then sem(b, r, s)
       else sem(c, r, s))
    else failwith ("nonboolean guard")
  | Val(e) -> match semden(e, r, s) with
    | Dloc n -> mvaltoeval(applystore(s, n))
    | _ -> failwith("not a variable")

and semden ((e:exp), (r:dval env), (s: mval store)) = match e with
  | Den(i) -> applyenv(r,i)
  | _ -> evaltodval(sem(e, r, s))

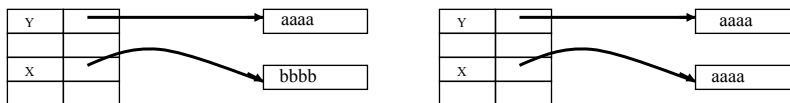
val sem : exp * dval Funenv.env * mval Funstore.store -> eval = <fun>
val semden : exp * dval Funenv.env * mval Funstore.store -> dval = <fun>

```

29

Semantica dell'assegnamento

- l'assegnamento coinvolge sia l'ambiente che la memoria
- vediamo il comportamento considerando l'assegnamento $X := Y$
 - `Assign(Den "X", Val "Y")`
 - dove sia X che Y sono variabili



- l'assegnamento fa "copiare" un valore nella memoria e non modifica l'ambiente
- quando i valori sono strutture dati modificabili
 - s-espressioni in LISP, arrays in ML, oggetti in Java
 il valore è in realtà un puntatore
 - le modifiche effettuate a partire da una variabile si ripercuotono sull'altra (sharing)
- scorciatoia in LISP e Java
- l'assegnamento agisce direttamente sull'ambiente e crea aliasing

30

Semantica operativa: comandi

```
let rec semc((c: com), (r:dval env), (s: mval store)) = match c with

| Assign(e1, e2) ->
  (match semden(e1, r, s) with
  | Dloc(n) -> update(s, n, evaltomval(sem(e2, r, s)))
  | _ -> failwith ("wrong location in assignment"))

| Cifthenelse(e, c11, c12) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semc(c11, r, s) else semc(c12, r, s))
  else failwith ("nonboolean guard")

| While(e, c1) -> let g = sem(e, r, s) in
  if typecheck("bool",g) then
    (if g = Bool(true) then semc((c1 @ [While(e, c1)]), r, s)
    else s)
  else failwith ("nonboolean guard")

and semcl(c1, r, s) = match c1 with
| [] -> s
| c::c11 -> semc(c11, r, semc(c, r, s))

val semc : com * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>
val semcl : com list * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>
```

31

Eliminare la ricorsione

- per le espressioni, bisogna prevedere il caso in cui il valore è un dval
 - nuova pila di valori denotabili temporanei
 - diverse etichette per le espressioni
- per i comandi, la ricorsione può essere rimpiazzata con l'iterazione senza utilizzare pile ulteriori
- il dominio dei comandi è "quasi" tail recursive

```
type com =
  | Assign of exp * exp
  | Cifthenelse of exp * com list * com list
  | While of exp * com list
```
- non è mai necessario valutare i due rami del condizionale
- si può utilizzare la struttura sintattica (lista di comandi) per mantenere l'informazione su quello che si deve ancora valutare
 - basta una unica cella
 - che possiamo "integrare" nella pila di espressioni etichettate
- il valore restituito dalla funzione di valutazione semantica dei comandi (uno store!) può essere gestito come aggiornamento di una "variabile globale" di tipo store

32

Le strutture dell'interprete iterativo

```
let cframesize = 20
let tframesize = 20
let tdframesize = 20

type labeledconstruct =
  | Expr1 of exp
  | Expr2 of exp
  | Exprd1 of exp
  | Exprd2 of exp
  | Com1 of com
  | Com2 of com
  | Coml of labeledconstruct list

let (continuation: labeledconstruct stack) = emptystack(cframesize, Expr1(Eint(0)))

let (tempstack: eval stack) = emptystack(tframesize, Novalue)

let (tempdstack: dval stack) = emptystack(tdframesize, Unbound)

let globalstore = ref(emptystore(Undefined))

let labelcom (dl: com list) = let dlr = ref(dl) in
  let ldir = ref([]) in
  while not (!dlr = []) do
    let i = List.hd !dlr in
    ldir := !ldir @ [Com1(i)];
    dlr := List.tl !dlr
  done;
  Coml(!ldlr)
```

33

L'interprete iterativo 1

```
let itsem (rho:dval env) =
  (match top(continuation) with
  | Expr1(x) ->
    (pop(continuation); push(Expr2(x), continuation);
    (match x with
    | Iszero(a) -> push(Expr1(a), continuation)
    | Eq(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Prod(a,b) -> push(Expr1(a), continuation); push(Expr1(b), continuation)
    | Sum(a,b) -> push(Expr1(a), continuation);
    push(Expr1(b), continuation)
    | Diff(a,b) -> push(Expr1(a), continuation);
    push(Expr1(b), continuation)
    | Minus(a) -> push(Expr1(a), continuation)
    | And(a,b) -> push(Expr1(a), continuation);
    push(Expr1(b), continuation)
    | Or(a,b) -> push(Expr1(a), continuation);
    push(Expr1(b), continuation)
    | Not(a) -> push(Expr1(a), continuation)
    | Ifthenelse(a,b,c) -> push(Expr1(a), continuation)
    | Val(a) -> push(Exprd1(a), continuation)
    | _ -> ()))
```

34

L'interprete iterativo 2

```
|Expr2(x) ->
  (pop(continuation); (match x with
    | Eint(n) -> push(Int(n),tempstack)
    | Ebool(b) -> push(Bool(b),tempstack)
    | Den(i) -> push(dvaltoeval(applyenv(rho,i)),tempstack)
    | Iszero(a) -> let arg=top(tempstack) in pop(tempstack); push(iszero(arg),tempstack)
    | Eq(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(equ(firstarg,sndarg),tempstack)
      | Prod(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(mult(firstarg,sndarg),tempstack)
    | Sum(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(plus(firstarg,sndarg),tempstack)
    | Diff(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(diff(firstarg,sndarg),tempstack)
    | Minus(a) -> let arg=top(tempstack) in pop(tempstack); push(minus(arg),tempstack)
    | And(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(et(firstarg,sndarg),tempstack)
    | Or(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
      let sndarg=top(tempstack) in pop(tempstack); push(vel(firstarg,sndarg),tempstack)
    | Not(a) -> let arg=top(tempstack) in pop(tempstack); push(non(arg),tempstack)
    | Ifthenelse(a,b,c) -> let arg=top(tempstack) in pop(tempstack);
      if typecheck("bool",arg) then
        (if arg = Bool(true) then push(Expr1(b),continuation)
         else push(Expr1(c),continuation))
      else failwith ("type error"))))
  | Val(e) -> let v = top(tempstack) in pop(tempstack);
  (match v with
    | Dloc n -> push(mvaltoeval(applystore(iglobalstore, n)), tempstack)
    | _ -> failwith("not a variable"))
  | _ -> failwith("no more cases for semexpr"))
val itsem : dval Funenv.env -> unit = <fun>
```

35

L'interprete iterativo 3

```
let itsemnden(rho) =
  (match top(continuation) with
  | Exprd1(x) -> (pop(continuation); push(Exprd2(x),continuation);
    match x with
      | Den i -> ()
      | _ -> push(Expr1(x), continuation))
  | Exprd2(x) -> (pop(continuation); match x with
    | Den i -> push(applyenv(rho,i), tempstack)
    | _ -> let arg = top(tempstack) in pop(tempstack);
      push(evaltodval(arg), tempstack))
  | _ -> failwith("No more cases for semden" ) )
val itsemnden : dval Funenv.env -> unit = <fun>
```

36

```

let itsemcl (rho: dval env) =
  let c1 = (match top(continuation) with
    | Com1(d11) -> d11
    | _ -> failwith("impossible in semdecl")) in
  if c1 = [] then pop(continuation) else
  (let currc = List.hd c1 in let newc1 = List.tl c1 in pop(continuation); push(Com1(newc1),continuation);
  (match currc with
    | Com1(Assign(e1, e2)) -> pop(continuation); push(Com1(Com2(Assign(e1, e2))::newc1),continuation);
    push(Exprd1(e1), continuation); push(Expr1(e2), continuation)
    | Com2(Assign(e1, e2)) -> let arg2 = evaltomval(top(tempstack)) in pop(tempstack);
    let arg1 = top(tempdstack) in pop(tempdstack); (match arg1 with
      | Dloc(n) -> globalstore := update(!globalstore, n, arg2)
      | _ -> failwith ("wrong location in assignment"))
    | Com1(While(e, c1)) -> pop(continuation); push(Com1(Com2(While(e, c1))::newc1),continuation);
    push(Expr1(e), continuation)
    | Com2(While(e, c1)) -> let g = top(tempstack) in pop(tempstack);
    if typecheck("bool",g) then (if g = Bool(true) then (let old = newc1 in let new1 =
      (match labelcom c1 with
        | Com1 new11 -> new11
        | _ -> failwith("impossible in while")) in
      let nuovo = Com1(new1 @ [Com1(While(e, c1))] @ old) in pop(continuation);
      push(nuovo,continuation))
    else ()) else failwith ("nonboolean guard")
    | Com1(Cifthenelse(e, c11, c12)) -> pop(continuation);
    push(Com1(Com2(Cifthenelse(e, c11, c12))::newc1),continuation); push(Expr1(e), continuation)
  )

```

37

```

|   if typecheck("bool",g) then (let temp = if g =
  Bool(true) then
    labelcom (c11) else labelcom (c12) in let
  new1 = (match temp with
    | Com1 new11 -> new11
    | _ -> failwith("impossible in
  cifthenelse")) in
    let nuovo = Com1(new1 @
  newc1Com2(Cifthenelse(e, c11, c12)) -> let g =
  top(tempstack) in pop(tempstack);
  ) in pop(continuation); push(nuovo,continuation)
    else failwith ("nonboolean guard")
    | _ -> failwith("no more sensible cases in
  commands" ) )
val itsemcl : dval Funenv.env -> unit = <fun>

```

38

```

let initState() = svuota(continuation); svuota(tempstack)
val initState : unit -> unit = <fun>

let loop (rho) =
  while not(empty(continuation)) do
    let currconstr = top(continuation) in (match currconstr with
      | Expr1(e) -> itsem(rho)
      | Expr2(e) -> itsem(rho)
      | Exprd1(e) -> itsemden(rho)
      | Exprd2(e) -> itsemden(rho)
      | Coml(cl) -> itsemcl(rho)
      | _ -> failwith("non legal construct in loop"))
    done
  val loop : dval Funenv.env -> unit = <fun>

let sem (e,(r: dval env), (s: mval store)) = initState();
  globalstore := s; push(Expr1(e), continuation);
  loop(r); let valore= top(tempstack) in pop(tempstack);
  valore
val sem : exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

let semden (e,(r: dval env), (s: mval store)) = initState();
  globalstore := s; push(Exprd1(e), continuation);
  loop(r); let valore= top(tempdstack) in pop(tempdstack);
  valore
val semden : exp * dval Funenv.env * mval Funstore.store -> dval = <fun>

let semcl (cl,(r: dval env), (s: mval store)) = initState();
  globalstore := s; push(labelcom(cl), continuation);
  loop(r); !globalstore
val semcl : com list * dval Funenv.env * mval Funstore.store -> mval Funstore.store = <fun>

```

39