

Implementazione di (ambiente e) memoria nel linguaggio imperativo

1

Contenuti

- ambiente e memoria locale nel linguaggio imperativo
 - cosa serve in ogni attivazione
 - perché la restrizione a locazioni denotabili
 - implementazione: strutture dati e operazioni
 - cosa cambia nell'interprete iterativo

2

Ambiente locale dinamico

- per ogni attivazione
 - entrata in un blocco o chiamata di proceduraabbiamo attualmente nel record di attivazione gli interi ambiente e memoria
 - implementati come funzioni
- in accordo con la semantica dell'ambiente locale dinamico possiamo inserire nel record di attivazione
 - una tabella che implementa il solo ambiente locale (catena statica)
 - una tabella che implementa la memoria locale
- quando l'attivazione termina
 - uscita dal blocco o ritorno della chiamata di procedurapossiamo eliminare l'ambiente e la memoria locali insieme a tutte le altre informazioni contenute nel record di attivazione
 - ma si può fare solo se si impongono opportune restrizioni

3

La pila di ambienti locali

- stessa soluzione del linguaggio funzionale con creazione di un ambiente locale nuovo per
 - `Let`, `Apply`, `Block`, `Call`
- insieme di pile di ambienti locali
 - `namestack`, pila di array di identificatori
 - `dvalstack`, pila di array di valori denotati
 - `slinkstack`, pila di (puntatori ad) ambienti
 - `tagstack`, pila di etichette per la retention
 - solo per il frammento funzionale
 - le procedure non sono esprimibili

4

E la memoria locale?

storestack, pila di `mval` store

- la creazione di una nuova memoria locale avviene chiaramente quando si entra in un blocco o si chiama una procedura
 - se ci sono dichiarazioni di variabile
 - creiamo una associazione tra un nome ed una espressione di tipo `Newloc`
 - un array di `mval`, con tante posizioni quante sono le dichiarazioni di variabile
- il nuovo storestack, pila di array di `mval`
- uno store è un puntatore nella pila (intero)
 - e lo store corrente è il valore della variabile `currentstore`
- una locazione è una coppia di interi:
 - il primo identifica lo store
 - il secondo la posizione relativa
- tutte le variabili raggiungibili attraverso l'ambiente non locale sono accessibili
 - possono essere lette
 - possono essere modificate con l'assegnamento
 - possono essere passate come parametri

5

Strutture dati e operazioni

```
type 't store = int
let (currentstore: mval store ref) = ref(0)
let storestack = emptystack(stacksize, [Undefined])
let (newloc, initloc) = let count = ref(-1) in
  (fun () -> count := !count + 1; (!currentstore, !count)),
  (fun () -> count := -1)
let applystore ((x: mval store), ((n1, n2): loc)) =
  let a = access(storestack, n1) in Array.get a n2
let emptystore(x) = initloc(); svuota(storestack); currentstore := -1; !currentstore
let allocate ((s:mval store), (m:mval)) = let (n1, n2) = newloc() in
  let a = access(storestack, n1) in Array.set a n2 m; ((n1, n2), s)
let update((s:mval store), (n1,n2), (m:mval)) =
  if applystore(s, (n1,n2)) = Undefined then failwith ("wrong assignment")
  else let a = access(storestack, n1) in Array.set a n2 m; s
let pushlocalstore (dl) = let rdl = ref(dl) in let rn = ref(0) in
  while not(!rdl = []) do
    let (i, d) = List.hd !rdl in
      (match d with | Newloc(_) -> rn := !rn + 1 | _ -> ());
      rdl := List.tl !rdl
  done; let a = Array.create !rn Undefined in pop(storestack); push(a, storestack);
  initloc(); !currentstore
```

- `pushlocalstore` ha come argomento una lista di dichiarazioni (costanti e variabili)
 - crea l'array locale della dimensione necessaria e lo sostituisce a quello (vuoto) correntemente sulla testa di `storestack`
- `allocate` setta l'array già così creato

6

Gestione a pila della memoria locale 1

- per poter correttamente poppare anche la memoria locale insieme al resto del record di attivazione dobbiamo essere sicuri che non esistano cammini d'accesso "esterni" alle locazioni interne alla memoria locale
 - diversi da quelli costituiti dall'ambiente locale
 - che stanno per essere eliminati
- un cammino d'accesso esterno può essere
 - un altro nome, diverso da quello locale, per la locazione (aliasing)
 - l'unico meccanismo che può crearlo è il passaggio della locazione come parametro (per riferimento!) ad altra procedura
 - ma, al momento del ritorno della procedura che conteneva la dichiarazione di variabile originale,
 - qualunque procedura chiamata è già necessariamente ritornata
 - l'aliasing non può esistere più
 - una locazione appartenente a diversa memoria locale che contiene la locazione come valore
 - impossibile perché le locazioni non sono valori memorizzabili
 - il valore temporaneo della attuale "applicazione di funzione"
 - impossibile perché le locazioni non sono valori esprimibili
- la memoria "dinamica" può essere gestita a pila solo se
 - le locazioni non sono nè esprimibili nè memorizzabili
 - le locazioni sono diverse dai puntatori ed i puntatori non esistono!

7

Gestione a pila della memoria locale 2

- digressione su casi non compresi nel linguaggio didattico
 - la gestione dinamica a pila della memoria è stata inventata da ALGOL 60
 - insieme all'ambiente locale dinamico ed alla struttura a blocchi
 - ALGOL 60 non prevede
 - nè puntatori
 - nè vere strutture dati dinamiche (vedi dopo)
 - se il linguaggio prevede i puntatori (PASCAL, C)
 - è necessaria una gestione della memoria a heap
 - simile a quella vista per le liste ed a quella che vedremo per gli oggetti
 - che può coesistere con una gestione a pila
 - se viene mantenuta una distinzione fra locazioni e puntatori (PASCAL)
 - una gestione dinamica della memoria può essere necessaria anche in linguaggi che non hanno nella semantica nè store nè heap
 - linguaggi funzionali, linguaggi logici
- semplicemente perché esistono vere strutture dati dinamiche
- liste, termini, s-espressioni
- implementate necessariamente con heap e puntatori
- tutti i casi che richiedono la gestione a heap
 - puntatori, strutture dinamiche, oggetti
- permettono che operazioni che richiedono l'allocazione dinamica di memoria
- new di strutture o di classi, applicazione di costruttori ricorsivi
- possano figurare in punti arbitrari del programma
- invece che soltanto nelle dichiarazioni all'ingresso dei blocchi!

8

Alcune novità nell'interprete iterativo 1

```
let applyproc ((ev1:dval),(ev2:dval list), s) = ( match ev1 with
| Dprocval(Proc(ii,(l1, l2, l3)), x) ->
    let r = bindlist(x, ii, ev2) in
    newframes(labelcom(l3), r, s);
    push(Rdecl(l2), top(cstack));
    push(labeldec(l1),top(cstack));
    pushlocalenv(l1,l2,r);
    pushlocalstore(l1); ()
| _ -> failwith ("attempt to apply a non-functional object"))
```

- **chiamata di procedura**

- un unico frame

- l'ambiente locale che contiene i parametri e lo "spazio" per tutte le dichiarazioni del blocco
- la memoria locale che contiene lo spazio per tutte le variabili locali

9

Alcune novità nell'interprete iterativo 2

```
let itsemrdecl() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let r1 = (match top(continuation) with
  | Rdecl(r11) -> r11
  | _ -> failwith("impossible in semrdecl")) in
  pop(continuation);
  let prl = ref(r1) in
  while not(!prl = []) do
    let currd = List.hd !prl in
    prl := List.tl !prl;
    let (i, den) =
      (match currd with
      |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rho))
      |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rho))
      | _ -> failwith("no more sensible cases in recursive declaration")) in
    currentenv := bind(rho, i, den)
  done
```

- **non c'è più il punto fisso**

- l'ambiente delle chiusure (rho) è lo stesso in cui vengono inserite le associazioni

10

Alcune novità nell'interprete iterativo 3

```
let semb ((l1, l2, l3), r, s) = initState(r,s);
    newframes(labelcom(l3), r, s);
    push(Rdecl(l2), top(cstack));
    push(labeldec(l1),top(cstack));
    pushlocalenv(l1,l2,!currentenv);
    pushlocalstore(l1);
    loop();
    currentenv := !currentenv + 1;
    currentstore := !currentstore + 1;
    topstore()
```