Analisi Statica, traduzione dei nomi

- Riferimenti:

  www.di.unipi.it/~basile

  Trovate il materiale di questa esercitazione e altre informazioni.

Analisi Statica: traduzione dei nomi

- dato il programma, per tradurre una specifica occorrenza di Den ide bisogna
  - identificare con precisione la struttura di annidamento
  - identificare il blocco o funzione dove occorre l'associazione per ide (o scoprire subito che non c'è) e vedere in che posizione è ide in tale ambiente
  - contare la differenza delle profondità di nesting

- un modo conveniente per ottenere questo risultato con una tecnica simile a quella che userà il valutatore parziale è costruire una analisi statica

- una esecuzione del programma con l'interprete appena definito che esegue solo i costrutti che hanno a che fare con l'ambiente (ignorando tutti gli altri) dell'ambiente guarda solo nomi e link statici (namestack e slinkstack)

- che costruisce un nuovo ambiente locale seguendo la struttura statica (Let, Fun, Rec) e non quella dinamica (Let e Apply) facendo attenzione ad associare ad ogni espressione l'ambiente in cui deve essere valutata

- chiaramente diverso dalla costruzione dell'ambiente a tempo di esecuzione basata sulle applicazioni e non sulle definizioni di funzione, ma sappiamo che per la traduzione dei nomi è la struttura statica quella che conta

Analisi Statica, Interprete

- Il vecchio interprete viene sostituito da un traduttore ed un interprete
- Il traduttore valuta parzialmente il programma in input, producendo un altro programma dove tutti I nomi sono sostituiti da offset nella catena statica
- Il nuovo interprete prende in input il codice valutato parzialmente, e accede direttamente alla pila di ambienti evalstack senza cercare il nome nella pila parallela namestack

Esempio: input traduttore

```
val expo : exp =

Let

    ("expo",

     Rec

       ("expo",

        Fun

          (["base"; "esp"],

           Let

             ("f",

              Fun

                (["x"],

                 Ifthenelse

                   (Iszero (Den "esp"),Appl (Den "f",[Eint 1]),

                    Prod

                      (Den "x",

                       Appl (Den "expo",[Den "x"; Diff (Den "esp",Eint 1)])))),

              Appl (Den "f",[Den "base"])))),

     Appl (Den "expo",[Den "x"; Eint 3]))
```

- Il traduttore prende in input una espressione
- Nomi e Den ide sono evidenziati in rosso

Esempio : output traduttore

- L'output del compilatore è una nuova espressione di tipo staticexp
- I nomi sono scomparsi
- Den ide sono stati tradotti in
  - Access(int,int) se il nome esisteva nella catena statica
  - Sunbound se il nome non era visibile nella catena statica

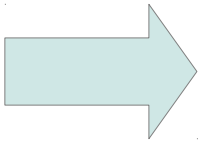
```
> val it : staticexp =
  SBind
    (SRec
       (SFun
          (SBind
             (SFun
                (Sifthenelse
                   (Siszero (Access (1,1)),SAppl (SUnbound,[SInt 1]),
                    Smult
                      (Access (0,0),
                       SAppl
                         (Access (2,0),
                          [Access (0,0); Sdiff (Access (1,1),SInt 1)])))),
              SAppl (Access (0,0),[Access (1,0)])))),
    SAppl (Access (0,0),[SUnbound; SInt 3]))
```

Domini : expr

- Nuovo tipo da exp a staticexp
- Bisogna tradurre i nomi in offset nella catena statica
- Den ide viene trasformato in Access of int*int
- Ide viene eliminato da tutti gli altri costrutti, in quanto verrà tradotto in offset

Domini: expr

```
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
  | Let of ide * exp * exp
  | Fun of ide list * exp
  | Appl of exp * exp list
  | Rec of ide * exp
```



```
type staticexp =
  | SUnbound
  | SInt of int
  | SBool of bool
  | Access of int * int
  | Smult of staticexp * staticexp
  | Splus of staticexp * staticexp
  | Sdiff of staticexp * staticexp
  | Sequ of staticexp * staticexp
  | Sminus of staticexp
  | Siszero of staticexp
  | Svel of staticexp * staticexp
  | Set of staticexp * staticexp
  | Snon of staticexp
  | Sifthenelse of staticexp * staticexp *
staticexp
  | SBind of staticexp * staticexp
  | SFun of staticexp
  | SAppl of staticexp * staticexp list
  | SRec of staticexp
```

Prima “fase” traduzione (1)

```
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) ->  (*staticamente eseguiamo entrambi i rami dell'if per tradurre i Den(i) interni*)
            push(Expr1(a),continuation)
            push(Expr1(b),continuation)
            push(Expr1(c),continuation)
        | Appl(a,b) ->
            push(Expr1(a),continuation)
            pushargs(b,continuation)
```

Prima fase traduzione (2)

```
| Let(i,e1,e2) -> push(Expr1(e1),continuation)
                  (* Prima di chiudere la funzione in una espressione statica bisogna
                     valutare il corpo in un nuovo frame*)
                  newframes(e2,bind(rho, i, SUnbound))
| Fun(i,a) ->
    let foo = List.init (List.length i) (fun _ -> Unbound)
                                  (* ignoriamo i valori dei parametri, usiamo una lista dummy*)
    newframes(a,bindlist(rho, i, foo))
                (* prima di chiudere la funzione in una espressione statica bisogna
                    valutare il corpo in un nuovo frame*)
| Rec(f,e) ->
    newframes(e,bind(rho, f, Unbound))    (*  i costrutti Rec sono sempre nella forma
                                               Let ( ide, (Rec ( ide, Fun(... ) ), ... )
                                               Bisogna creare un nuovo frame con il binding della funzione ricorsiva,
                                               Successivamente il costrutto Fun costruira' un nuovo frame contenente i
                                               parametri della funzione.
                                               Prima di fare la chiusura per il costrutto Rec (verra' fatto nel ramo
                                               Exp2) bisogna valutare il corpo per effettuare eventuali traduzioni.
                                                          *)
```

```
let newframes(e,rho) =
  let cframe = emptystack(cframesize(e),Expr1(e))
  let tframe = emptystack(tframesize(e),SUnbound)
  push(Expr1(e),cframe)
  push(cframe,cstack)
  push(tframe,tempsexpstack)
  pushenv(rho)
```

- E' necessario ricostruire la struttura degli ambienti tramite newframes per tradurre correttamente i Den Ide in accessi

Prima fase traduzione (3)

```
| Let(i,e1,e2) -> push(Expr1(e1),continuation)
                  newframes(e2,bind(rho, i, SUnbound))
```

- A tempo di traduzione non abbiamo bisogno dei valori memorizzati nella pila evalstack
- Nell'interprete vecchio si poteva costruire il nuovo frame con il binding solo dopo aver valutato Expr(e1), cioè nella seconda fase della traduzione
- Noi anticipiamo alla prima fase la creazione del frame, inserendo nel campo valore un SUnbound

Seconda “fase” traduzione (1)

Adesso tempstack è una pila di staticexp

```
| Expr2(x) ->
          pop(continuation)
          match x with
            | Eint(n) ->  push(SInt(n),tempstack)
            | Ebool(b) -> push(SBool(b),tempstack)
            | Den(i) ->   push(applyenv(rho,i),tempstack) (* qui verra' tradotto in Access(int,int)*)
            | Iszero(a) ->
                let arg=top(tempstack)
                pop(tempstack)
                push(Siszero(arg),tempstack)
            | Eq(a,b) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Sequ(firstarg,sndarg),tempstack)
            | Prod(a,b) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Smult(firstarg,sndarg),tempstack)
            | Sum(a,b) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Splus(firstarg,sndarg),tempstack)
            | Diff(a,b)  ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Sdiff(firstarg,sndarg),tempstack)
            | Minus(a) ->
                let arg=top(tempstack)
                pop(tempstack)
                push(Sminus(arg),tempstack)
            | And(a,b) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Set(firstarg,sndarg),tempstack)
            | Or(a,b) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                push(Svel(firstarg,sndarg),tempstack)
            | Not(a) ->
                let arg=top(tempstack)
                pop(tempstack)
                push(Snon(arg),tempstack)
            | Ifthenelse(a,b,c) ->
                let firstarg=top(tempstack)
                pop(tempstack)
                let sndarg=top(tempstack)
                pop(tempstack)
                let trdarg=top(tempstack)
                pop(tempstack)
                push(Sifthenelse(firstarg,sndarg,trdarg),tempstack)
```

Seconda “fase” traduzione (2)

```
| Den(i) ->   push(applyenv(rho,i),tempstack)
```

```
let applyenv ((x: evalenv), (y: ide)) =
  let n = ref(x)
  let found = ref(false)
  let accesses = ref 0
  let index = ref 0
  while not !found && !n > -1 do
    let lenv = access(namestack,!n)
    let nl = Array.length lenv
    index := 0
    while not !found && !index < nl do
      if Array.get lenv !index = y then
        found := true
      else
        index := !index + 1
    done
    if not !found then
      n := access(slinkstack,!n)
      accesses := !accesses + 1
  done
  if !found then
    Access(!accesses, !index)
  else
    SUnbound
```


- La applyenv del vecchio interprete restituiva il valore alla locazione di ide
- Nel nostro caso, ci interessa memorizzare i due offset accesses e index
- Non restituiremo un eval ma una staticexp Access(x,y)
- Dato che non interessa il valore memorizzato, la pila evalstack risulta inutile
- Restituiamo una chiusura lessicale contenente gli offset nella pila degli ambiente per prelevare il valore a runtime
- Il nuovo interprete non avrà bisogno di effettuare la ricerca per nome, la pila namestack diventerà inutile

Seconda fase traduzione (3)

```
| Fun(i,a) ->
    let arg=top(tempstack)
    pop(tempstack)
    push(SFun(arg),tempstack)
| Rec(f,e) ->
    let arg=top(tempstack)
    pop(tempstack)
    push(SRec(arg),tempstack)
| Let(i,e1,e2) ->
    let firstarg=top(tempstack)
    pop(tempstack)
    let sndarg=top(tempstack)
    pop(tempstack)
    push(SBind(firstarg, sndarg),tempstack)
| Appl(a,b) ->
    let firstarg=top(tempstack)
    pop(tempstack)
    let sndarg=getargs(b,tempstack)
    push(SAppl(firstarg,sndarg),tempstack)
```

- Una volta tradotte le sottoespressioni in ambienti con i binding effettuati correttamente, è possibile chiudere lessicalmente i costrutti

Terminazione traduzione

```
    ...
  done
  let valore= top(top(tempsexpstack))
  pop(top(tempsexpstack))
  pop(cstack)
  pop(tempsexpstack)
  push(valore,top(tempsexpstack))
  popenv()
done
let valore = top(top(tempsexpstack))
pop(tempsexpstack)
valore
```

- Al termine della traduzione, come nel vecchio interprete, viene restituito il valore presente nella pila temporanea
- Nel nostro caso non sarà un eval
- Ma una espressione statica contenente tutto l'albero di sintassi astratta del programma in input
- Dove avremo tradotto tutti i costrutti in costrutti alternativi senza ide
- E tutti i Den(ide) in Access(int,int)
- Il codice prodotto può essere adesso dato in pasto al nuovo interprete
- E la retention?

Nessuna Retention

- E' necessario fare Retention quando una espressione valuta un Funval(Fun(ii,aa),r)
- Se facciamo popenv l'ambiente r non esisterà più al momento dell'applicazione, e il valore relativo di slinkstack sarà compromesso

```
# sem(
   Appl(
     Appl(
       Fun(["x"],
          Fun(["y"],Sum(Den("x"),Den("y")))),
       [Eint 3]),
     [Eint 5])),
emptyenv Unbound);;
```

- Durante l'analisi statica ci interessa solamente tradurre i nomi
- Quando incontriamo un Fun(i,exp) pushiamo un frame per exp e valuteremo eventuali identificatori interni, esempio: `Fun(["y"],Sum(Den("x"),Den("y"))`
- Quando applicheremo la funzione costruiremo semplicemente una chiusura lessicale: Sappl(firstarg,sndarg)
- L'argomento di Sappl sono le espressoni statiche con i nomi tradotti
- Non applichiamo mai, non avremo nessun problema di retention

Interprete

- Il nuovo interprete prenderà in input valori di tipo staticexp e valuterà ad eval
- Bisogna “duplicare” alcuni metodi usati dal traduttore in quanto cambiano i tipi:
  - Vecchio interprete: exp → eval
  - Traduttore: exp → staticexp
  - Nuovo interprete: staticexp → eval
- Alternativamente potevamo utilizzare funzioni polimorfe
- Scelta la soluzione più “semplice”:
  - Due pile cstack e tempstack
    - Cstack di tipo exp nel traduttore e staticexp nell'interprete
    - Tempstack di tipo staticexp nel traduttore e eval nell'interprete
  - Duplicati i metodi bind,bindlist,pushargs,getargs
    - Pushargs e getargs prendono exp nel traduttore e staticexp nell'interprete
    - Bind e bindlist hanno un numero diverso di parametri:
      - Nel traduttore abbiamo anche gli identificatori, che vengono eliminati nell'interprete
- Il dominio efun adesso chiude staticexp

Interprete

Traduttore

```
let bind ((r:evalenv),i,d)

let bindlist(r, il, el)

let pushargs ((b: exp
list),continuation) =

let getargs ((b: exp list),(tempstack:
staticexp stack)) =
```

Interprete

```
let bind2 ((r:evalenv),d) =

let bindlist2(r, el) =

let pushargs2 ((b: staticexp
list),continuation) =

let getargs2 ((b: staticexp list),
(tempstack: eval stack)) =
```

Domini: efun

- Adesso efun utilizzerà staticexp per le chiusure lessicali

```
| SFun(a) → push(makefun(SFun(a),rho),tempstack)
```

```
let makefun ((a:staticexp),(x:evalenv)) =
  match a with
    | SFun(aa) -> Funval(a,x)
    | _ -> failwith ("Non-functional
object")
```

```
   and eval =
        | Int of int
        | Bool of bool
        | Unbound
        | Funval of efun
```

```
    and efun = staticexp * (evalenv)
```

Prima fase interpretazione

```
match top(continuation) with
       | SExpr1(x) ->
           pop(continuation)
           push(SExpr2(x),continuation)
           match x with
             | Siszero(a) -> push(SExpr1(a),continuation)
             | Sequ(a,b) ->
                 push(SExpr1(a),continuation)
                 push(SExpr1(b),continuation)
             | Smult(a,b) ->
               push(SExpr1(a),continuation)
               push(SExpr1(b),continuation)
             | Splus(a,b) ->
               push(SExpr1(a),continuation)
               push(SExpr1(b),continuation)
             | Sdiff(a,b) ->
                 push(SExpr1(a),continuation)
                 push(SExpr1(b),continuation)
             | Sminus(a) -> push(SExpr1(a),continuation)
             | Set(a,b) ->
               push(SExpr1(a),continuation)
               push(SExpr1(b),continuation)
             | Svel(a,b) ->
                 push(SExpr1(a),continuation)
                 push(SExpr1(b),continuation)
             | Snon(a) -> push(SExpr1(a),continuation)
             | Sifthenelse(a,b,c) -> push(SExpr1(a),continuation)
             | SBind(e1,e2) -> push(SExpr1(e1),continuation)
             | SAppl(a,b) ->
                 push(SExpr1(a),continuation)
                 pushargs2(b,continuation)
             | _ -> ()
```

- Non cambia niente rispetto al vecchio interprete
- Abbiamo solamente staticexp invece di exp

Seconda fase interpretazione(1)

```
| SExpr2(x) ->
    pop(continuation)
    match x with
      | SUnbound -> push(Unbound,tempstack)
      | SInt(n) ->  push(Int(n),tempstack)
      | SBool(b) -> push(Bool(b),tempstack)
      | Siszero(a) ->
          let arg=top(tempstack)
          pop(tempstack)
          push(iszero(arg),tempstack)
      | Sequ(a,b) ->
          let firstarg=top(tempstack)
          pop(tempstack)
          let sndarg=top(tempstack)
          pop(tempstack)
          push(equ(firstarg,sndarg),tempstack)
      | Smult(a,b) ->
          let firstarg=top(tempstack)
          pop(tempstack)
          let sndarg=top(tempstack)
          pop(tempstack)
          push(mult(firstarg,sndarg),tempstack)
      | Splus(a,b) ->
          let firstarg=top(tempstack)
          pop(tempstack)
          let sndarg=top(tempstack)
          pop(tempstack)
          push(plus(firstarg,sndarg),tempstack)
      | Sdiff(a,b)  ->
          let firstarg=top(tempstack)
          pop(tempstack)
          let sndarg=top(tempstack)
          pop(tempstack)
          push(diff(firstarg,sndarg),tempstack)
      | Sminus(a) ->
          let arg=top(tempstack)
          pop(tempstack)
          push(minus(arg),tempstack)
      | Set(a,b) ->
          let firstarg=top(tempstack)
          pop(tempstack)
          let sndarg=top(tempstack)
          pop(tempstack)
          push(et(firstarg,sndarg),tempstack)
```

```
| Svel(a,b) ->
    let firstarg=top(tempstack)
    pop(tempstack)
    let sndarg=top(tempstack)
    pop(tempstack)
    push(vel(firstarg,sndarg),tempstack)
| Snon(a) ->
    let arg=top(tempstack)
    pop(tempstack)
    push(non(arg),tempstack)
| Sifthenelse(a,b,c) ->
    let arg=top(tempstack)
    pop(tempstack)
    match arg with
      | Bool(bg) ->
          if bg  then
            push(SExpr1(b),continuation)
          else
            push(SExpr1(c),continuation)
      | _ -> failwith ("type error")
```

Fino a qui tutto come il vecchio interprete ad eccezione delle staticexp

Seconda fase interpretazione (2)

```
| Access(l,i) ->   push(accessenv(rho,l,i),tempstack)
```

```
let accessenv ((x: evalenv), (l: int), (index: int)) =
  let l = ref(l)
  let n = ref(x)
  while !l > 0 do
    n := access(slinkstack,!n)
    l := !l - 1
  Array.get (access(evalstack,!n)) index
```

- Adesso non abbiamo Den(ide) ma Access(int,int), non eseguiamo più una ricerca del nome nella pila degli ambienti, ma accediamo direttamente
- Abbiamo eliminato un ciclo ed un confronto tra stringhe ad ogni iterazione del ciclo, risparmiando anche nelle strutture dati (la pila namestack è inutile)
- Scorriamo la catena statica con slinkstack fino all'ambiente dove si trova il valore che cerchiamo
- Dopo con Array.get restituiamo l'eval che corrispondeva all'ide che abbiamo tradotto

Seconda fase interpretazione (3)

```
| SFun(a) ->
     push(makefun(SFun(a),rho),tempstack)
| SRec(e) -> push(makefunrec(e,rho),tempstack)
| SBind(e1,e2) ->
    let arg=top(tempstack)
    pop(tempstack)
    newframes(e2,bind2(rho, arg))
| SAppl(a,b) ->
    let firstarg=top(tempstack)
    pop(tempstack)
    let sndarg=getargs2(b,tempstack)
    applyfun(firstarg,sndarg)
```

```
let makefun ((a:staticexp),(x:evalenv)) =
  match a with
    | SFun(aa) -> Funval(a,x)
    | _ -> failwith ("Non-functional object")

let applyfun ((ev1:eval),(ev2:eval list)) =
  match ev1 with
    | Funval(SFun(aa),r) -> newframes(aa,bindlist2(r, ev2))
    | _ -> failwith ("…")

let makefunrec ((a:staticexp),(x:evalenv)) =
  makefun(a, ((lungh(namestack) + 1): evalenv))
```

- Rispetto al vecchio interprete makefun applyfun e makefunrec prendono delle staticexp invece di exp
- Sono scomparsi gli ide