Tecniche per il passaggio dei parametri

Contenuti

- la tecnica base (nei vari paradigmi)
  - passaggio per costante, per riferimento, di funzioni, procedure e oggetti
- altre tecniche
  - passaggio per nome
    - nuovi costrutti
    - nuovi domini
    - semantica operazionale dei nuovi costrutti
  - argomenti funzionali à la LISP (solo scoping dinamico)
  - passaggio per valore, per risultato e per valore-risultato
    - confronto tra passaggio per riferimento e per valore-risultato

Passaggio dei parametri

- due meccanismi per condividere informazione fra sottoprogrammi diversi
  - associazione non locale o globale
    - quando l'entità da condividere è sempre la stessa
  - parametro
    - quando l'entità da condividere cambia da attivazione ad attivazione
- argomenti formali
  - lista dei nomi locali usati per riferire dati non locali
- argomenti attuali
  - lista di espressioni, i cui valori saranno condivisi
- tra argomenti formali ed attuali c'è una corrispondenza posizionale
  - stesso numero (e tipo) degli argomenti nelle due liste
    - in certi linguaggi i parametri attuali possono essere di meno
- cosa è il passaggio dei parametri?
- binding, uno alla volta, nell'ambiente ρ tra il parametro formale (locale) e il valore denotato ottenuto dalla valutazione dall'argomento attuale
  - la regola di scoping influenza l'identità dell'ambiente (non locale) ρ, ma non l'effetto su ρ del passaggio di parametri
  - l'associazione per un nome locale viene creata dal passaggio invece che da una dichiarazione

La tecnica base di passaggio

- per funzioni, procedure, classi
  - vediamo, per semplicità, solo il caso delle procedure nel linguaggio imperativo

```
type proc = exp *  dval env

let rec semden ((e:exp), (r:dval env), (s: mval store)) =  match e with
  | Proc(i,b) -> (Dprocval(Proc(i,b), r), s)

let rec semc ((c: com), (r:dval env), (s: mval store)) = match c with
  | Call(e1, e2) -> let (p, s1) = semden(e1, r, s)  in
       let (v, s2) = semlist(e2, r, s1) in
        match p with   | Dprocval(Proc(i,b), x), -> semb(b, bindlist (x, i, v), s)
```

- si riduce a
  - valutazione della lista di argomenti nello stato corrente
    - che produce una lista di `dval` (`eval` nel linguaggio funzionale)
  - costruzione di un nuovo ambiente legando ogni identificatore della lista dei formali `x` con il corrispondente valore della lista di `dval v`
    - `identificatore` e `valore` devono avere lo stesso tipo

Che valori possono essere passati?

```
type dval =
          | Dint of int
          | Dbool of bool
          | Dloc of loc
          | Dobject of pointer
          | Dfunval of efun
          | Dprocval of proc
```

- a seconda del tipo del valore che viene passato si ottengono varie modalità note
  - tutte semanticamente equivalenti
    - per costante
    - per riferimento
    - di oggetti
    - argomenti procedurali
  - in tutte parametro formale ed espressione corrispondente hanno lo stesso tipo

Passaggio per costante

```
type dval =
          | Dint of int
          | Dbool of bool
          | Dloc of loc
          | Dobject of pointer
          | Dfunval of efun
          | Dprocval of proc
```

- "`x`" ha come tipo un valore tradizionale non modificabile
- l'espressione corrispondente valuta ad un `Dval` rosso
  - l'oggetto denotato da "`x`" non può essere modificato dal sottoprogramma
  - il passaggio per costante esiste in alcuni linguaggi imperativi e in tutti i linguaggi funzionali
    - anche il passaggio ottenuto via pattern matching e unificazione è quasi sempre un passaggio per costante

Passaggio per riferimento

```
type dval =
            | Dint of int
            | Dbool of bool
            | Dloc of loc
            | Dobject of pointer
            | Dfunval of efun
            | Dprocval of proc
```

- "`x`" ha come tipo un valore modificabile (locazione)
- l'espressione corrispondente valuta ad un `Dloc`
  - l'oggetto denotato da "`x`" può essere modificato dal sottoprogramma
- crea aliasing
  - il parametro formale è un nuovo nome per una locazione che già esiste
- e produce effetti laterali
  - le modifiche fatte attraverso il parametro formale si ripercuotono all'esterno
- il passaggio per riferimento esiste in quasi tutti i linguaggi imperativi

Passaggio di oggetti

```
type dval =
            | Dint of int
            | Dbool of bool
            | Dloc of loc
            | Dobject of pointer
            | Dfunval of efun
            | Dprocval of proc
```

- "`x`" ha come tipo un puntatore ad un oggetto
- l'espressione corrispondente valuta ad un `Dobject`
  - il valore denotato da "`x`" non può essere modificato dal sottoprogramma
  - ma l'oggetto da lui puntato può essere modificato

Passaggio di funzioni, procedure (e classi)

```
type dval =
            | Dint of int
            | Dbool of bool
            | Dloc of loc
            | Dfunval of efun
            | Dprocval of proc
```

- "`x`" ha come tipo una funzione, una procedura o una classe
- l'espressione corrispondente
  - nome di procedura o classe, anche espressione che ritorna una funzione

  valuta ad un `dval` rosso
  - l'oggetto denotato da "`x`" è una chiusura
  - può solo essere ulteriormente passato come parametro o essere attivato (`Apply`, `Call`, `New`)
  - in ogni caso il valore ha tutta l'informazione che serve per valutare correttamente l'attivazione
- nei linguaggi imperativi e orientati a oggetti di solito anche le funzioni non sono esprimibili
- in LISP (linguaggio funzionale con scoping dinamico) gli argomenti funzionali si passano in un modo più complesso (vedi dopo)

Altre tecniche di passaggio

- in aggiunta al meccanismo base visto, esistono altre tecniche di passaggio dei parametri che
  - non coinvolgono solo l'ambiente
    - i passaggi per valore e risultato coinvolgono anche la memoria
  - non valutano il parametro attuale
    - passaggio per nome
  - cambiano il tipo del valore passato
    - argomenti funzionali in LISP

Passaggio per nome

- l'espressione passata in corrispondenza di un parametro per nome `"x"`
  - non viene valutata al momento del passaggio
  - ogni volta che (eventualmente) si incontra una occorrenza del parametro formale `"x"`
  - l'espressione passata ad `"x"` viene valutata
- perché?
- per definire (funzioni e) sottoprogrammi *non-stretti* su uno (o più di uno) dei loro argomenti
  - come nella regola di valutazione esterna delle espressioni
  - l'attivazione può dare un risultato definito anche se l'espressione, se valutata, darebbe un valore indefinito (errore, eccezione, non terminazione)
    - semplicemente perché in una particolare esecuzione `"x"` non viene mai incontrato

Passaggio per nome nei linguaggi imperativi

- dato che l'espressione si valuta ogni volta che si incontra il parametro formale
  - nella memoria corrente

  il passaggio per nome e per costante possono avere semantiche diverse anche nei sottoprogrammi stretti
  - se l'espressione contiene variabili che possono essere modificate (come non locali) dal sottoprogramma
- diverse occorrenze del parametro formale possono dare valori diversi

Passaggio per nome in semantica operazionale

- l'espressione passata in corrispondenza di un parametro per nome "`x`"
  - non viene valutata al momento del passaggio
  - viene (eventualmente) valutata ogni volta che si incontra una occorrenza del parametro formale "`x`" dal punto della semantica denotazionale
- una espressione non valutata è una chiusura

  `exp * dval env`
- la valutazione dell'occorrenza di "`x`" si effettua
  - valutando la chiusura denotata da "`x`"
    - l'espressione della chiusura, nell'ambiente della chiusura e nella memoria corrente
- anche in un linguaggio funzionale puro una espressione non valutata è una chiusura

  `exp * eval env`
- la valutazione dell'occorrenza di "`x`" si effettua valutando l'espressione della chiusura nell'ambiente della chiusura

Passaggio per nome: semantica operazionale

```
type exp = ....
    | Namexp of exp
    | Namden of ide
type dval = Unbound
    | Dint of int
    | Dbool of bool
    | Dloc of loc
    | Dfunval of efun
    | Dprocval of proc
    | Dnameval of namexp
and namexp = exp * dval env
let rec sem ((e:exp), (r:dval env), (s: mval store)) = match e with
   | ....
   | Nameden(i) -> match applyenv(r,i) with
         Dnameval(e1, r1) -> sem(e1, r1, s)
and semden ((e:exp), (r:dval env), (s: mval store)) = match e with
   | Namexp e1 -> (Dnameval(e1,r), s)
   | .....

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

Espressioni per nome e funzioni

- una espressione passata per nome è chiaramente simile alla definizione di una funzione (senza parametri)
  - che “si applica” ogni volta che si incontra una occorrenza del parametro formale
- stessa soluzione semantica delle funzioni
  - chiusura in semantica operazionale (e nelle implementazioni)
- l’ambiente che viene fissato (nella chiusura) è quello di passaggio
  - che, per le espressioni, è l’equivalente della definizione
- mentre la semantica delle funzioni è influenzata dalla regola di scoping, ciò non è vero per le espressioni passate per nome
  - che vengono comunque valutate nell’ambiente di passaggio, anche con lo scoping dinamico
- il passaggio per nome è previsto in nobili linguaggi come ALGOL e LISP
  - è alla base dei meccanismi di valutazione lazy di linguaggi funzionali moderni come Haskell
  - può essere simulato in ML semplicemente passando funzioni senza argomenti!

Argomenti funzionali à la LISP

- come è noto, LISP ha lo scoping dinamico
  - il dominio delle funzioni in semantica operazionale è

```
type efun = exp
```

- un argomento formale "`x`" di tipo funzionale dovrebbe denotare un `eval` della forma `Funval(efun)`
- se "`x`" viene successivamente applicato, il suo ambiente di valutazione dovrebbe correttamente essere quello del momento dell'applicazione
- la semantica di LISP prevede invece che l'ambiente dell'argomento funzionale venga congelato nel momento del passaggio
- questo porta alla necessità di introdurre due diversi domini per funzioni e argomenti funzionali

```
type funarg = exp * eval env
```

- i domini degli argomenti funzionali sono chiaramente identici ai domini delle funzioni con scoping statico
  - ma l'ambiente rilevante (quello della chiusura) è quello del passaggio e non quello di definizione

Argomenti funzionali à la LISP 2

- quando una funzione viene passata come argomento ad altra funzione in una applicazione
  - passando un nome di funzione, una lambda astrazione o una qualunque espressione la cui valutazione restituisca una funzione
- la funzione deve essere prima di tutto "chiusa" con l'ambiente corrente ρ
  - inserendo ρ nella chiusura

  da `efun = exp` a

  `funarg = exp * eval env`
- nelle implementazioni coesisteranno funzioni rappresentate dal solo codice ed argomenti funzionali rappresentati da chiusure (codice, ambiente)
  - con un numero notevole di complicazioni associate
  - un esempio subito

Ritorno di funzioni à la LISP

- quando una applicazione di funzione restituisce una funzione, il valore restituito dovrebbe essere di tipo `efun`
- in alcune implementazioni di LISP si segue la medesima strada degli argomenti funzionali
  - viene restitito un valore di tipo `funarg`
- anche questa scelta complica notevolmente l'implementazione
  - come vedremo quando parleremo di implementazione dell'ambiente
  - si hanno gli stessi problemi dello scoping statico (retention), senza averne i vantaggi (verificabilità, ottimizzazioni)
    - perché tanto odio?

Chiusure per tutti i gusti

- nelle semantiche operazionali e nelle implementazioni un'unica soluzione per
  - espressioni passate per nome (con tutte e due le regole di scoping)
  - funzioni, procedure e classi con scoping statico
  - argomenti funzionali e ritorni funzionali con scoping dinamico à la LISP

Passaggio per valore

- non la semantica formale, ma solo l'intuizione
  - ha a che fare con i valori modificabili e non esiste quindi nei linguaggi funzionali
- nel passaggio per valore il parametro attuale è un valore di tipo `t`, mentre il parametro formale `"x"` è una variabile di tipo `t`
  - argomento e parametro formale hanno tipi diversi e non possiamo fare il passaggio con una operazione di `bind` nell'ambiente
  - di fatto `"x"` è il nome di una variabile locale alla procedura, che, semanticamente, viene creata prima del passaggio
  - il passaggio diventa quindi un assegnamento del valore dell'argomento alla locazione denotata dal parametro formale
    - coinvolge la memoria e non l'ambiente, se l'assegnamento è implementato correttamente
    - non viene creato aliasing e non ci sono effetti laterali anche se il valore denotato dal parametro formale è modificabile
      - a differenza di ciò che accade nel passaggio per costante
    - permette il passaggio di informazione solo dal chiamante al chiamato

Passaggio per valore-risultato

- per trasmettere anche informazione all'indietro dal sottoprogramma chiamato al chiamante, senza ricorrere agli effetti laterali diretti del passaggio per riferimento
- sia il paramentro formale "`x`" che il parametro attuale "`y`" sono variabili di tipo `t`
- "`x`" è una variabile locale del sottoprogramma chiamato
- al momento della chiamata del sottoprogramma, viene effettuato un passaggio per valore (`x := !y`)
  - il valore della locazione (esterna) denotata da "`y`" è copiato nella locazione (locale) denotata da "`x`"
- al momento del ritorno dal sottoprogramma, si effettua l'assegnamento inverso (`y := !x`)
  - il valore della locazione (locale) denotata da "`x`" è copiato nella locazione (esterna) denotata da "`y`"
- esiste anche il passaggio per risultato solo

Valore-risultato e riferimento

- il passaggio per valore risultato ha un effetto simile a quello del passaggio per riferimento
  - trasmissione di informazione nei due sensi tra chiamante e chiamato

  senza creare aliasing
  - la variabile locale contiene al momento della chiamata una copia del valore della variabile non locale
  - durante l'esecuzione del corpo della procedura le due variabili denotano locazioni distinte
    - e possono evolvere indipendentemente
  - solo al momento del ritorno la variabile non locale riceve il valore da quella locale
- nel passaggio per riferimento, invece, viene creato aliasing e i due nomi denotano esattamente la stessa locazione
- i due meccanismi sono chiaramente semanticamente non equivalenti
  - per mostrarlo basta considerare una procedura la cui semantica dipenda dal valore corrente della variabile non locale `"y"`
    - nel passaggio per riferimento, ogni volta che modifico la variabile locale, modifico anche `"y"`
    - nel passaggio per risultato, `"y"` viene modificato solo alla fine

Passaggio dei parametri in LISP e Java

- ricordiamo che l'assegnamento in LISP e Java è realizzato
  - senza copiare (nella memoria)
  - ma inserendo nell'ambiente i puntatori (alla S-espressione o all'oggetto)
- la stessa cosa succede nel passaggio di parametri, che, quindi,
  - può essere spiegato in termini di assegnamento
  - può essere chiamato per valore