AA 2015-2016

26. Il passaggio dei parametri

Condivisione dei binding

- **Associazione non locale o globale**
  - comodo quando quando l'entità da condividere è sempre la stessa
- **Parametri**
  - importante 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
  - **Formali-Attuali**: corrispondenza posizionale
    - ✓ stesso numero (e tipo) degli argomenti nelle due liste

Cosa è il passaggio dei parametri?

- Una specia di ***dichiarazione dinamica***
  - binding nell'ambiente 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)
  - l'associazione per un nome locale viene creata dal passaggio invece che da una dichiarazione

Procedure


```
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)
```

- 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 i 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 passato si ottengono varie modalità note
  - per costante
  - per riferimento
  - di oggetti
  - argomenti procedurali
- In tutte le modalità il parametro formale e l'espressione attuale corrispondente devono avere lo stesso tipo

- Valori di default
  - In alcuni linguaggi (ad esempio Python, Ruby, PHP) i parametri formali possono assumere dei valori di default nel caso il parametro attuale sia assente

- Numero variabile di parametri
  - Dal manuale di Python: “the actual is a list of values and the corresponding formal parameter is a name with an asterisk”

Passaggio per costante


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

- Il parametro formale “x” ha come tipo un valore non modificabile
- L’espressione corrispondente valuta a un Dval (in grassetto)
- 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
```

- Il parametro formale “x” ha come tipo un valore modificabile (locazione)
- L’espressione corrispondente valuta a un Dloc
  - l’oggetto denotato da “x” può essere modificato dal sottoprogramma
- Crea aliasing e produce effetti laterali
  - il parametro formale è un nuovo nome per una locazione che già esiste
  - 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
```

- Il parametro formale “x” ha come tipo un puntatore a un oggetto
- L’espressione corrispondente valuta a 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 e procedure (e classi)


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

- Il parametro formale "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
  - 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 neanche le funzioni sono esprimibili
- In LISP (linguaggio funzionale con scoping dinamico) gli argomenti funzionali si passano in un modo più complesso

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 valore

- Meccanismo che coinvolge i valori modificabili e non esiste quindi nei linguaggi funzionali puri con dati immutabili
  - si parla di passaggio per costante
- Nel passaggio per valore il parametro attuale è un valore di tipo t, il parametro formale "x" una variabile di tipo t
  - 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

Passaggio per valore

- 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 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 le due variabili denotano locazioni distinte
  - e possono evolvere indipendentemente
  - solo al momento del ritorno la variabile non locale riceve il valore dalla 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

Linguaggi e parametri

- C
  - Pass-by-value
  - Pass-by-reference con puntatori
- C++
  - Puntatore (reference type) per pass-by-reference
- Java
  - Tutti i parametri sono trasmessi by value
  - Oggetti by reference

Linguaggi e parametri

- Fortran 95+
  - in, out, o inout
- C#
  - Default: pass-by-value
    - Pass-by-reference inserendo ref
- PHP: +-= a C#
- Perl: array predefinito @_

Type checking

- FORTRAN 77 e C: non lo avevano
- Pascal, FORTRAN 90+, Java e Ada: si
- ANSI C e C++: prototype
- Perl, JavaScript, e PHP non richiedono type checking
- Python and Ruby: type checking non è possibile

Un caso di studio: passaggio per nome

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 a "x" viene valutata
- Per definire (funzioni e) sottoprogrammi non stretti su uno (o più di uno) dei loro argomenti
  - 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: semantica

- 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"
- 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


```
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 passando funzioni senza argomenti!

Argomenti funzionali à la LISP

- LISP ha lo scoping dinamico, pertanto il dominio delle funzioni 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 a un'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 r
  - inserendo r nella chiusura

  da **efun = exp** a **funarg = exp * eval env**
- Nelle implementazioni coesisteranno funzioni rappresentate dal solo codice e argomenti funzionali rappresentati da chiusure (codice, ambiente)
  - con un numero notevole di complicazioni associate

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
  - si hanno gli stessi problemi dello scoping statico (retention), senza averne i vantaggi (verificabilità, ottimizzazioni)

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