Il passaggio dei parametri

1

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

2

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

3

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
/* dval tipi denotabili */
```

- 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

4

Altre considerazioni

- **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”

5

Esempio: Python

```
def welcome(name,msg):
    print("Ciao",name + ', ' + msg)

welcome("Pippo", "Buon Pomeriggio!")
```

```
>>> welcome("Pippo") # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'
```

6

Python

```
def wecome(name, msg = "Buona Giornata!"):

  print("Ciao",name + ', ' + msg)

welcome("Pluto")
welcome("Topolino","How do you do?")
```

```
Ciao, Pluto, Buona Giornata!
Ciao,Topolino, How do you do?
```

7

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.
- 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

8

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
- Una qualche forma di passaggio per riferimento esiste in molti linguaggi, anche se spesso è "simulato" (vedi i puntatori in C)

9

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

10

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 le funzioni non sono esprimibili

11

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

12

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

13

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

14

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

15

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

16

Linguaggi e parametri

- C
  - Pass-by-value
- Java
  - Pass-by-value

Un caso di studio: passaggio per nome (nei linguaggi funzionali)

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)

19

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

20

Passaggio per nome: semantica

```
type exp = ...
   | Namexp of exp
   | Namden of ide
type evT = Unbound
    | Int of int
    | Bool of bool
    | Nameval of namexp
  and namexp = exp * evT env
```

21

Interprete

```
let rec eval ((e: exp), (r: evT env) =
  match e with
        :
        | Namexp e1 -> (Nameval(e1, r)
        | Nameden(i) -> match applyenv(r, i) with
              Nameval(e1, r1) -> eval(e1, r1)
              | fail …….
```

22

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 del chiamante (l'equivalente della definizione) ma 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!

23

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)

24

Esercizio

```
int z = 1;
void p (int x) < body >;
p(z);
print(z)
```

Scrivere il corpo di p in modo tale che la chiamata si comporti differentemente in caso di
pass by value,
pass by reference,
pass by value-result.

25

Esercizio

```
int z = 1;
void p (int x) < body >;
p(z)
print(z)
```

x=x+1 ; z = z+2

Scrivere il corpo di p in modo tale che la chiamata si comporti differentemente in caso di
pass by value -- 3
pass by reference -- 4
pass by value-result – 2

26

Call_by_value vs by reference vs value_result

```
int n;
void  p(int k); {n = n+1; k = k+4; print(n);}
main{n = 0; p(n); print(n);}

Risultato
call by value: 1 1
call by value-result: 1 4
call by reference: 5 5
```

call by reference versus call by name

```
int a[1..4];
int n;
void p(int b){
print(b); n = n+1; print(b); b = b+5;}
main{
a[1] = 10;
a[2] = 20;
a[3] = 30;
a[4] = 40;
n := 1;
p(a[n+2]);
print(a);
}
call by reference: 30 30 10 20 35 40
call by name: 30 40 10 20 30 45
```

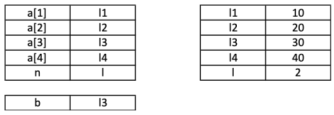
ambiente

a[1]	l1		l1	10
a[2]	l2		l2	20
a[3]	l3		l3	30
a[4]	l4		l4	40
n	l		l	1
b	l3			

memoria

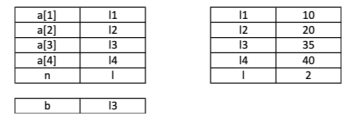
call by reference versus call by name

```
int a[1..4];
int n;
void p(int b){
print(b); n = n+1; print(b); b = b+5;}
main{
a[1] = 10;
a[2] = 20;
a[3] = 30;
a[4] = 40;
n := 1;
p(a[n+2]);
print(a);
}
call by reference: 30 30 10 20 35 40
call by name: 30 40 10 20 30 45
```

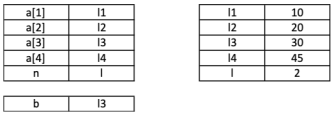


call by reference versus call by name

```
int a[1..4];
int n;
void p(int b){
print(b); n = n+1; print(b); b = b+5;}
main{
a[1] = 10;
a[2] = 20;
a[3] = 30;
a[4] = 40;
n := 1;
p(a[n+2]);
print(a);
}
call by reference: 30 30 10 20 35 40
call by name: 30 40 10 20 30 45
```



Call by reference



Call by name