Progetto di Linguaggi

Nota sul trattamento delle funzioni ricorsive.

Nel linguaggio LispKit le funzioni ricorsive possono venire introdotte solo dal costrutto Letrec associato alla costrutto Lambda. Questo può avvenire con un binder come questo:

a=Lambda(x,y).... a(..)....; in cui il corpo della funzione chiama il nome a della parte sinistra del binder. E' anche possibile definire vari binder che si invocano mutuamente, come in:

a=Lambda(x,y).... b(..)....;

b=Lambda(x).... a(..)....;

Il valore di una funzione è sempre una chiusura (C,E) dove C è il corpo e E l'ambiente (della SECD machine) al momento in cui la funzione è definita. Nelle funzioni (sia ricorsive che non), l'ambiente in cui la funzione è eseguita (cioè dopo che Ap viene eseguita) è: A E, dove A è la lista dei parametri attuali ed E l'ambiente contenuto nella chiusura. Nel caso di un Letrec che definisce nei binder funzioni ricorsive, il corpo C del Letrec deve essere eseguito in un ambiente B E, dove B è la lista dei valori delle parti destre dei binder. Quindi tra questi valori ci saranno anche le chiusure delle funzioni ricorsive definite dai binder. E è invece l'ambiente all'inizio dell'esecuzione del Letrec che serve per trovare i valori globali sia delle parti destre dei binder che del corpo della Letrec. Ora assumiamo che un binder definisca la funzione a() che è ricorsiva in modo diretto, cioè essa invoca se stessa. Il caso di funzioni mutuamente ricorsive è un pò più complicato, ma può essere spiegato in modo analogo. Assumiamo inoltre che la chiusura corrispondente ad a() sia (C',E'). Da quanto detto prima, B deve contenere (C',E'). Cosa sia C' è ovvio: il corpo di a(). Meno chiaro è invece cosa sia E'. Vogliamo mostrare che E'=B E e che quindi E' contiene (C',E') , insomma è una struttura circolare.

La circolarità di E' è necessaria, infatti la funzione a() è ricorsiva e quindi C' conterrà un'invocazione di a(). Quando la corrispondente Ap viene eseguita l'ambiente deve essere A E' (dove A sono i parametri attuali) e quindi il valore (C',E') dovrà essere in E' che quindi deve essere circolare. Visto che E'=B E sappiamo anche che la circolarità sta in B infatti ogni chiusura (C',E') in B contiene se stessa in E'. B sembra più difficile da capire che da costruire. Nel programma interprete.hs, B è costruito dalle funzioni lazyE e lazyClo partendo dalla lista L dei valori delle parti destre dei binder. In corrispondenza della funzione ricorsiva a(), L conterrà la chiusura (C',E) dove E è l'ambiente (in linea di principio non circolare) corrente al momento in cui viene creata la chiusura stessa (attraverso l'esecuzione di un Ldf(C')). lazyE e lazyClo prendono L e trasformano ciascuna chiusura (C',E) di L in (C', B E). La lista L così trasformata è B.

Per finire consideriamo il caso in cui tra i binder di un Letrec ci siano binder che non definiscono funzioni ricorsive (o mutuamente ricorsive). Per esempio un binder come a=x+2. In questo caso il valore di x andrebbe cercato nell'ambiente corrente quando x+2 viene valutato. Ma quando x+2 viene compilato, l'ambiente statico usato è (NB) (NE), dove NB è la lista delle parti sinistre dei binder e NE l'ambiente statico all'inizio della compilazione del Letrec. L'iniziale N indica che si tratta di liste di Nomi. Insomma l'ambiente statico ha un RA in più rispetto a quello dinamico e quindi l'indirizzo per x che il compilatore calcola rischia di essere sbagliato quando viene usato per la valutazione del binder. Si risolve il problema aggiungendo un RA fittizio ([OGA]) in cima all'ambiente dinamico usato per la valutazione di ciascuna parte destra di binder.

Nel caso dei binder ricorsivi [OGA] viene sostituito dalla lista circolare B, mentre per i binder non ricorsivi resta [OGA] che ha solo il compito di far funzionare gli indirizzi calcolati dal compilatore (come quello di x in x+2). E' l'istruzione Push della SECD machine che si occupa di mettere [OGA] in cima all'ambiente dinamico all'inizio di un Letrec. Il compilatore deve inserire Push all'inizio del codice che produce per ogni Letrec. Nel programma interprete.hs dato, se in un letrec vengono inseriti vari binder non ricorsivi, come a=x+2 e b=a+3,..., l'a che appare nella parte destra del secondo binder non verrà associato al primo binder, bensì verrà cercato nell'ambiente all'inizio del letrec. Se invece ci sono questi 2 binder: a=x+2 e f = lambda() ...a... allora la a che appare nel corpo della funzione (sia che f sia ricorsiva sia che non lo sia) viene associato al primo binder. L'ordine tra i 2 binder non è influente per questo fenomeno. Il seguente programma LispKit permette di osservare quanto appena descritto:

h="let x = let y = 3 in letrec x = y+1 and f = lambda(z) z+x in f end end in x(2) end $"

In questo programma si verifica anche l'upward-fun-result problem: f definita nel letrec annidato diventa il valore di x del let più esterno. A questo punto il letrec interno ha terminato la sua esecuzione avendo restituito il suo risultato f. Dovrebbe quindi scomparire portando via anche la variabile locale x (attenzione la x del letrec) che è invece usata da f che viene eseguita quando si invoca x(2). Nella nostra macchina SECD, anziché copiare i RA nello heap, essi sono nella chiusura di f: la chiusura di f ha nella sua seconda componente B E (usando la terminologia introdotta prima), la lista dei binder B (resa circolare) che contiene il valore di questo x usato da f. Insomma la chiusura di f (e di tutte le funzioni) nella SECD è assolutamente autonoma, non ha bisogno di alcuna informazione esterna per venire eseguita.

In conclusione va anche osservato che Ap viene usato per ogni invocazione di funzione sia ricorsiva che non ricorsiva, mentre Rap viene usato solo nei Letrec cioè là dove le funzioni ricorsive vengono definite. Rap invoca le funzioni lazyE e lazyClo che preparano la seconda componente circolare dentro alle chiusure delle funzioni definite nei binder del letrec. Attenzione che questo succede sia che le funzioni definite nel letrec siano ricorsive, sia che non lo siano.