Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

IL CONTROLLO DEGLI EFFETTI LATERALI IN ALCUNI LINGUAGGI PER LA PROGRAMMAZIONE SEQUENZIALE E CONCORRENTE

S. Gnesi

Nota interna B85-14

Ottobre 1985

IL CONTROLLO DEGLI EFFETTI LATERALI IN ALCUNI LINGUAGGI PER LA PROGRAMMAZIONE SEQUENZIALE E CONCORRENTE

S. Gnesi, Istituto di Elaborazione dell'Informazione, C.N.R. Pisa

1. INTRODUZIONE

Un obiettivo prioritario nella progettazione dei linguaggi di programmazione e' quello della protezione, ma una delle maggiori limitazioni ad una soluzione sicura di tali problemi e' data dalla presenza di effetti laterali; infatti la presenza di questi e' un ostacolo per la comprensione dei programmi. Inoltre gli errori dovuti agli effetti laterali sono piu' difficili da scoprire, cosa resa possibile solo esaminando attentamente tutti i meccanismi del linguaggio e delle loro possibili interazioni.

In generale, i linguaggi di programmazione usati piu' comunemente permettono la presenza di "side-effects". Comunque alcuni dei linguaggi piu' nuovi come EUCLID / 7,8 /, ALPHARD / 9 / hanno affrontato questo problema fornendone delle interessanti soluzioni.

Nel prossimo capitolo diamo una rassegna delle principali fonti di effetti laterali non controllati cosi' come sono permessi nei linguaggi di programmazione piu' diffusi.

Nel capitolo 3 affronteremo il problema dell'aliasing, mostrando come un meccanismo di scoping assicuri che questa non venga creata sia in ambienti sequenziali che concorrenti.

Nel capitolo 4 mostriamo come si puo' ottenere un ulteriore controllo degli effetti laterali associando ad

ogni identificatore un attributo che specifica come l'oggetto denotato da tale identificatore possa essere usato.

Infine nel capitolo 5 si presentano alcuni linguaggi di programmazione che permettono l'uso di processi concorrenti.

2. EFFETTI LATERALI

Si hanno effetti laterali quando l'esecuzione di una istruzione provoca un effetto nascosto accanto a quello principale. Il problema nasce principalmente dalle chiamate di procedure e funzioni, in quanto, in generale, l'esecuzione di una istruzione di chiamata puo' modificare oggetti oltre quelli esplicitamente denotati nell'istruzione stessa.

Le principali ragioni per cui effetti laterali vengono generati sono:

a) variabili globali nascoste;

b) procedure con variabili statiche;

c) sharing;

d) aliasing.

Le variabili globali nascoste sono i globali di una

procedura che non sono noti nell'ambiente della chiamata.

Un caso in cui si possono avere globali nascosti deriva dalla possibilita' di ridichiarare, in linguaggi tipo Algol, identificatori in un blocco piu' interno e anche a causa delle possibilita' di passare procedure come parametri.

Un simile comportamento si verifica nei linguaggi i cui programmi sono costituiti da moduli separati con meccanismi di "imports" ed "exports". Infatti si potrebbe dichiarare in un modulo globali e procedure che usano questi globali, e poi esportare dal modulo solo le procedure, per esempio:

```
module 1 (implementatore)          module 2 (utente )

  export P,Q;                        import P,Q;
  x: t
  ...                                ...
  procedure P ... end;               P
  procedure Q ... end;               Q
end                                end
```

dove le procedure P e Q usano la variabile globale x.

Le variabili locali statiche , conservando il loro valore da una chiamata all'altra, influenzano il valore

delle chiamate delle procedure, quindi due chiamate diverse di queste con parametri identici possono produrre effetti diversi. Le procedure con variabili locali statiche possono essere considerate come un caso speciale di procedure con variabili globali nascoste.

Anche lo sharing, che si verifica quando una variabile e' condivisa da piu' di un oggetto, e' una delle cause di effetti laterali. Questo accade in tutti i linguaggi che permettono l'uso di puntatori, in questo caso una modifica di un oggetto ne puo' provocare altre sugli oggetti che condividono la stessa variabile.

L'aliasing si verifica quando alla stessa variabile sono associati due nomi diversi nello stesso ambiente oppure nel caso di ambienti concorrenti, quando ci si puo' riferire con piu' nomi ad un oggetto, anche in ambienti diversi. Il caso piu' comune si ha in tutti i linguaggi che permettono il passaggio dei parametri "by reference" (PL /1, Pascal, FORTRAN) quando la chiamata di una procedura contiene piu' di una volta la stessa variabile.

3. ALIASING

Abbiamo visto che una fonte di "side effect" e' l'aliasing, cioe' quando due o piu' identificatori denotano lo stesso oggetto nel medesimo ambiente.

Per esempio si ha aliasing in tutti quei linguaggi, come PL/1 o Pascal, che permettono il passaggio dei parametri "by reference", cioe' quando lo stesso parametro e' passato due volte in una stessa chiamata, p. e. P (x, x).

Il concetto di aliasing e' importante, infatti il passaggio dei parametri a delle procedure puo' essere piu' facilmente descritto in termine dei legami tra gli identificatori di parametri formali e gli oggetti passati, come in Algol 68 / 1 /. In questo approccio, cosi', il passaggio dei parametri non comporta una memorizzazione o delle operazioni sui dati, ma piu' semplicemente modifica soltanto l'ambiente. Questo coincide con il passaggio di parametri "by reference". Inoltre, / 2 /, i legami degli identificatori nelle dichiarazioni possono essere trattati con lo stesso metodo. Da questo punto di vista il legame di un identificatore e' un operazione che crea aliasing, dato che essa da' un nuovo nome a un vecchio oggetto.

Quello che ci proponiamo di fare e' vedere come sia possibile controllare l'aliasing. L'idea e' quella di introdurre un meccanismo di scoping che assicuri che l'aliasing non sia creata. Risolviamo questo problema prima in un ambiente sequenziale poi in uno concorrente.

3.1 AMBIENTE SEQUENZIALE

In un ambiente sequenziale abbiamo aliasing quando ci si puo' riferire allo stesso oggetto con piu' di un nome nello stesso ambiente.

Introduciamo ora il meccanismo di scoping detto sopra. Assumiamo per ora che oggetti di tipo modificabile, cioe' oggetti che possono essere aggiornati senza perdere la propria identita', possono essere denotati soltanto da identificatori. Allora una dichiarazione dovrebbe essere :

y : t is x

dove x e' un identificatore conosciuto nel vecchio ambiente.

Dato un blocco:

```
begin
    y : t  is  x;
    ...
end
```

noi proponiamo che il nuovo ambiente sia ottenuto dal vecchio non solo legando y all'oggetto denotato da x, ma anche eliminando l'identificatore x stesso. In altre parole tutti gli identificatori conosciuti nel blocco piu' esterno sono importati nel blocco piu' interno, tranne quello usato per definire il nuovo identificatore.

Una sequenza di dichiarazioni e' trattata considerando ogni dichiarazione come appartenente ad un nuovo blocco annidato. Per esempio:

```
begin
    y : t  is  x;
    z : t  is  x;
    ...
end
```

e' equivalente a:

```
begin
    y : t  is  x;
    begin
        z : t  is  x;
        ...
    end
    ...
end
```

e cosi' la seconda dichiarazione non va bene, dato che x non e' conosciuto.

Immaginiamo ora di legare un nuovo identificatore a un oggetto ritornato da un'espressione. Consideriamo il seguente esempio:

```
begin
    y : t  is  x;
    z : t' is  f(x);
    ...
end
```

dove f e' una funzione che ritorna un nuovo oggetto. Valutando queste dichiarazioni, a causa delle regole dette sopra, noi diamo errore nel cercare di valutare f(x), dato che x e' stato eliminato dalla dichiarazione precedente. Se noi riusciamo a calcolare f(x) il legame di questo valore a z non causera' aliasing. Questo potrebbe essere risolto scambiando semplicemente le due dichiarazioni. Infatti avendo:

```
begin
    z : t' is  f(x);
    y : t  is  x;
    ...
end
```

il primo legame non elimina x, e cosi' il secondo puo' essere fatto.

3.2 AMBIENTE CONCORRENTE

Abbiamo visto fino ad ora come veniva affrontato il problema dell'alisiasing in ambienti sequenziali, in generale pero', dobbiamo risolverlo anche nel caso in cui si abbia una esecuzione concorrente di piu' processi. In tal caso abbiamo aliasing quando ci si puo' riferire allo stesso oggetto con piu' di un nome anche in ambienti diversi.

In generale i costrutti per realizzare programmazione concorrente sono del tipo, / 6 /:

```
cobegin
    P1 ( );
    P2 ( );
    ...
    Pn ( );
coend
```

dove P1, P2,...., Pn sono le procedure che possono essere eseguite in parallelo. Queste procedure, naturalmente potranno avere variabili condivise, usate per esempio per scambi di informazioni tra queste, es. buffers.

Un esempio tipico di programmazione concorrente si ha con la realizzazione di un sistema per la gestione di un problema produttore-consumatore di una risorsa, dove questa

e' l'oggetto condiviso dai due processi.

Torniamo ora al problema dell'aliasing in un ambiente concorrente.

Supponiamo di avere il seguente segmento di programma:

```
begin
    x : t1  is ...
    y : t2  is ...
    z : t3  is ...
    ...
    cobegin
        P1 (x,y);
        P2 ( z')
    coend
    ...
end
```

dove P1 e P2 sono due processi concorrenti.

Nel caso in cui sia x, y e z sono variabili non condivise da P1 e P2, la soluzione che si propone per controllare l'aliasing e' la stessa che nel caso sequenziale, cioe' stabilire una regola di scope per nascondere tutti gli identificatori conosciuti nel blocco piu' esterno tranne quello usato per definire il nuovo identificatore. In questo modo nel caso in cui si abbia:

```
begin
    x : t1  is ...
    y : t2  is ...
    ...
    cobegin
       P1 (x,y);
       P2 ( x )
    coend
    ...
end
```

al momento in cui avviene l'analisi dei parametri di P2 viene dato errore poiche' l'identificatore x non e' piu' conosciuto.

Questa soluzione va bene nel caso in cui i processi paralleli non comunichino tra loro con variabili condivise, ma questo non e' in realta' quello che piu' comunemente avviene. Per esempio nel caso del problema produttore-consumatore avremo:

```
begin
    BUFFER : t
    ...
    cobegin
       producer (BUFFER);
       consumer (BUFFER)
    coend
    ...
end
```

dove BUFFER e' condivisa tra i processi producer e

consumer. Applicando la regola di scope data sopra avremo che il processo consumer non conoscerebbe la variabile BUFFER. Questo, naturalmente, non ci va bene, quindi bisognera' in tal caso modificare la regola data, poiche' questa, in pratica, non permetterebbe l'uso di variabili condivise.

Una soluzione possibile, e' quella di escludere dal meccanismo di scope le variabili "shared". Per far questo si potrebbe associare ad ogni identificatore che denota questi oggetti un attributo che specifichi il suo stato:

attributo "shared" : l' oggetto e' condiviso da piu' di una procedura.

Quindi se in un certo ambiente un identificatore ha attributo "shared" questo non sara' nascosto nel nuovo ambiente.

Prendendo l'esempio visto sopra, il segmento di programma verra' cosi' modificato:

```
begin
    BUFFER : t shared is ...
    ...
    cobegin
        producer (BUFFER)
        consumer (BUFFER)
    coend
    ...
end
```

Quando verranno trattati i parametri del processo consumer, non avremo piu' errore dato che l'identificatore BUFFER ha ora associato l'attributo "shared", e quindi non e' nascosto nel nuovo ambiente.

Nel caso in cui i parametri di un processo contengono variabili non condivise con altri, su queste si applichera' la regola di scope data, cioe' esse saranno nascoste nell'ambiente piu' interno.

4. IDENTIFICATORI E ATTRIBUTI

Quanto e' stato visto fino ad ora riguarda oggetti modificabili, infatti applicare quanto detto a quelli non modificabili sarebbe eccessivamente restrittivo, dato che, per esempio, non potremmo permettere l'espressione x+x, con x intero, poiche' passando due volte lo stesso oggetto come parametro si crea aliasing.

Come abbiamo gia' detto gli oggetti modificabili sono oggetti che hanno uno stato che puo' essere modificato da alcune operazioni. Per fare una distinzione chiara tra oggetti modificabili, e non, si propone che quelli modificabili primitivi abbiano tipo var(t), dove t puo' essere qualunque tipo, e corrispondono alle variabili usate normalmente nei linguaggi di programmazione. Un oggetto di

tipo var(t) puo' essere modifcato da una operazione di assegnamento (assign) che assegna un oggetto di tipo t ad un oggetto di tipo var (t). Ad esempio si puo' avere :

```
x : integer is 5
y : var(integer) is allocate(integer,5);
assign (y,val(y)+x );
```

dove "allocate" e' una funzione che crea una variabile di tipo t inizializzata a 5 e val e' la funzione che restituisce il valore di una variabile.

Da questo esempio si puo' vedere che x e' legato per tutta la sua vita all'oggetto non modificabile 5 mentre y puo' essere modificato da operazioni di "assign".

Tornando al nostro problema anche questa distinzione non serve a risolverlo in ogni caso, infatti, anche riferendosi ad oggetti modificabili, puo' darsi che in certi ambienti si voglia usare un certo oggetto senza modificarlo, ed in questo caso l'aliasing non provocherebbe inconvenienti.

Per trattare questi casi proponiamo di associare un attributo ad ogni identificatore che specifichi il modo in cui si intende usare l'oggetto da esso denotato. In generale, gli oggettti strutturati possono avere sia parti modificabili che no, come per esempio:

```
z : record
      f1 : integer
      f2 : var(integer)
    end.
```

Gli attributi che proponiamo di associare agli identificatori sono i seguenti:

nessun attributo : si puo' accedere solo alle parti non modificabili dell'oggetto denotato come ad esempio z.f1

attributo "read" : si puo' accedere anche alle parti modificabili ma senza la possibilita' di modificarle.

attributo "write" : si puo' modificare l'oggetto. Si puo' per esempio assegnare un nuovo valore a z.f2.

Si puo' affermare che, se un identificatore ha in un certo ambiente attributo "write", questo non puo' essere in aliasing con nessun altro attributo "write" o "read".

Se tutte le funzioni restituiscono oggetti nuovi, l'unica situazione in cui si puo' avere aliasing e' quella in cui l'espressione dopo l'is e' costituita da un solo identificatore, come per esempio:

```
x : t  is  y
```

Infatti in questo caso, x verrebbe legato allo stesso

oggetto a cui e' legato y, mentre se dopo l'is ci fosse una generica espressione questa genererebbe un oggetto nuovo.

Per assicurare un uso corretto degli attributi, si deve verificare che l'attibuto associato ad x sia meno generale di quello associato ad y, mentre per evitare che l'aliasing avvenga causando "side effect", l'identificatore y deve essere importato cambiando il suo attributo se occorre.

riassumiamo tutto questo in una tabella:

	no attr.	read	write
no attr.	no attr.	read	write
read	-	read	write
write	-	-	no attr.

Le righe di questa tabella corrispondono agli attributi possibili del nuovo identificatore, le colonne ai possibili attributi di y e la tabella da' il suo nuovo attributo. Il trattino denota che non e' possibile effettuare il legame.

La modifica all'attributo di y rimane valida per tutto il tempo in cui esiste x.

Si puo' notare che un oggetto ora puo' essere denotato da piu' di un identificatore nello stesso ambiente ma, se uno di loro ha attributo "write", tutti gli altri non devono avere nessun attributo. Si puo' quindi dire che non si ha aliasing, dato che ogni identificatore puo' essere visto come legato ad oggetti diversi.

Queste regole per cio' che e' stato visto in precedenza valgono anche per il passaggio dei parametri. Applicando quanto detto e' ora possibile calcolare x+x. Infatti assumeremo che i parametri formali di tutte le operazioni su interi non abbiano attributi, dato che gli interi sono oggetti non modificabili.

Nel capitolo precedente abbiamo introdotto l'attributo

"shared", mentre in questo abbiamo parlato di associare agli identificatori o nessun attributo oppure quello di "read" o "write". Tra questi ultimi abbiamo anche stabilito una regola di "inclusione", ci si potrebbe quindi chiedere che relazione esiste tra l'attributo "shared" e gli altri. Si puo' affermare che questi ultimi non sono in alcun modo collegati, cioe' sono concetti tra loro mutuamente escludibiliMmmm, quindi ad oggetti "shared" non potremo anche associare l'attributo "read" o "write", e cosi' vale per il viceversa.

Fino ad ora abbiamo considerato solo il caso in cui si possano fornire solo operazioni di selezione su dati primitivi, come array o record, ma si potrebbe avere anche che le funzioni possano restituire una parte di un loro parametro. La valutazione di una generica espressione potra' restiture o un oggetto nuovo o una parte di un oggetto denotato da uno degli identificatori che compaiono nell'espressione stessa. Per poter trattare anche questo caso proponiamo che, se una funzione restituisce una parte di un suo parametro, questa informazione sia presente anche nel tipo. Ad esempio

```
func( x : t1,t2 ) ->  t3 part of x
```

e' il tipo di una funzione che restituisce un oggetto di

tipo t3 che e' parte del suo primo parametro.

Data una dichiarazione nella sua forma generale, conoscendo i tipi di tutte le funzioni usate e' possibile sapere se l'espressione dopo l'is restituisce parte di un oggetto. Le regole sui legami espresse nella tabella vista sopra si applicano ora a questo nome.

Ad esempio, si considerino gli oggetti a,x,y,f,p con i seguenti tipi:

```
a : var( integer ) write
x : var( integer ) write
y : var( integer ) read
f : func( var( integer ) write,
          n : var( integer))
           -> var( integer ) part of n
p : proc( var( integer ) read,
          var( integer ) write,
          var( integer ) read, var( integer ).
```

La seguente chiamata e' corretta:

```
p ( a,x,f( a,a ), y ).
```

Infatti per prima cosa vengono valutati i parametri attuali di p; la funzione f riceve due volte come parametro a, una volta con attributo "write" e una volta senza

attributo, e quindi la chiamata e' corretta. Questa chiamata restituisce una parte di a; valutiamo ora la chiamata di p legando i parametri nell'ordine. Dopo aver legato il primo parametro, l'attributo di a e' diventato "read"; dopo aver legato il secondo parametro, x non ha piu' attributi. Il terzo parametro attuale, come abbiamo visto, e' parte di a, che ha attualmente attributo "read", quindi anche questo legame puo' essere fatto ed a conserva lo stesso attributo. Infine viene legato l'ultimo parametro.

5. GESTIONE DELLA CONCORRENZA IN ALCUNI LINGUAGGI

Nel capitolo 3 abbiamo parlato di costrutti per gestione del parallelismo, vediamo ora come alcuni linguaggi di programmazione trattano questo problema.

Vediamo innanzitutto come la concorrenza e' gestita nel Pascal concorrente / 3 / tramite un esempio. Un caso tipico e' quello di un sistema in cui ci sono tre componenti. La prima un processo che produce un dato la seconda che lo usa, l'ultima un buffer che collega i due processi. Il costrutto di linguaggio che indica come un dato "shared" puo' essere usato da un processo e' il monitor. Esso puo' sincronizzare processi concorrenti e

trasmettere dati tra loro. Un monitor definisce una struttura dati condivisa e tutte le operazioni che possono essere realizzate su questa.

Diamo ora un esempio di programma concorrente in Concurrent Pascal:

```
type
    buffer = monitor ... end;
    inputprocess = process(buff:buffer);... end;
    outprocess   = process(buff:buffer);... end;
    var
       buffer1 : buffer;
       reader  : inputprocess;
       writer  : outprocess;
    begin
       init buffer1,
       reader ( buffer1 ),
       writer ( buffer1 )
    end
```

Da questo esempio si puo' vedere che le strutture dati condivise sono definite in modo esplicito nel testo del programma e quindi possono essere controllate dal compilatore, dando la possibilita' di compiere un'analisi completamente statica.

Anche il MESA / 4, 5 /, fornisce un supporto del linguaggio per l'esecuzione concorrente di piu' processi. Questo permette a programmi che sono inerentemente paralleli di essere chiaramente espressi. A differenza di quanto visto nel Concurrent Pascal, cioe' una soluzione del problema completamente strutturata e quindi staticamente verificabile, nel MESA il meccanismo proposto e' totalmente dinamico.

L'esecuzione parallela di due procedure e' permessa dagli statements di FORK e JOIN. Diamo un esempio di uso di FORK e JOIN. Consideriamo una procedura che compone interattivamente e edita linee di input:

```
readline:PROCEDURE s:STRING RETURN CARDINAL =
    BEGIN
    ...
    END;
```

Se questa procedura viene chiamata con il modo normale

```
n <- readline  buffer
```

il chiamante non puo' far niente altro durante l'elaborazione della stringa. Se c'e' qualche altra operazione che puo' essere fatta concorrentemente con

questa si puo' invece di chiamare readline fare un'operazione di FORK su questa:

```
p <- FORK readline buffer ;
   ...
   < computazione concorrente >
   ...
n <- JOIN p;
```

cio' permette agli statements in < computazione concorrente > di essere eseguiti in parallelo.

Dopo lo statement di JOIN, che viene eseguito solo dopo che tutti gli altri sono stati terminati, riprende l'esecuzione sequenziale. In generale, quando due o piu' processi sono cooperanti, essi hanno la necessita' di interagire in modo piu' complicato rispetto al FORK e JOIN. Il meccanismo per la sincronizzazione tra processi e' nel MESA una variante del monitor (Hoare, Brinch Hansen, Dijkstra). Il punto di vista e' che l'interazione tra processi si riduca sempre ad accessi sincronizzati ad oggetti condivsi.

Un modo di gestire il parallelismo piuttosto diverso da quello visto fino ad ora e' quello dell'ADA / 10 /. In questo linguaggio le entita' che possono operare in parallelo sono i "tasks". Il solo mezzo di comunicazione

tra questi e' tramite "entries", cioe' i tasks possono avere entries che possono essere chiamate da altri tasks. La sincronizzazione e' raggiunta con il "randezvous" tra il task che ha l'entry call e quello che accetta la chiamata.

Abbiamo visto che sia in Pascal Concorrente che in MESA le variabili condivise sono denotate da un costrutto del linguaggio, cioe' il monitor, in ADA non c'e' questa necessita' dal momento in cui tutto viene realizzato con i tasks.

Vediamo ora come e' realizzato in ADA il problema produttore-consumatore;
il task produttore conterra', p.e., i seguenti statement:

```
loop
 ... si produce il prossimo carattere CHAR
 BUFFER.WRITE(char);
 exit when CHAR=END-OF-TRASMISSION;
end loop;
```

il task consumatore conterra':

```
loop
  BUFFER.READ(CHAR)
    ... consuma il carrattere CHAR
  exit when CHAR=END-OF-TRASMISSION;     end loop;
```

il task che realizza il buffer sara':

```
task buffer is
   entry READ (c:out CHARACTER);
   entry WRITE(c:in  CHARACTER);
end.
```

Il task body BUFFER specifichera' come questo e' gestito in input ed output.

Tornando al problema dell'aliasing la situazione presentata per il suo controllo in ambienti concorrenti si avvicina al Pascal concorrente, mentre per il momento non si e' approfondito il caso di ADA, che prevede una gestione del parallelismo totalmente diversa.

BIBLIOGRAFIA

1. Lindsey,c.h.,van der Mulen,s.g., Informal Introduction to Algol 68, North Holland, 1973.

2. Tennent,R.D.,Language Design Methods Based on Semantic Principles, Acta Informatica 8, 1977.

3. Brinch Hansen,P., The Programming Language Concurrent Pascal, IEEE Trasaction on Software Engineering, Vol SE-1, no. 2, 1975.

4. Mitchell,J.G.et al.,Mesa Language Manual, Version 5.0 CSL-79-3,XEROX-PARC, 1979.

5. Lampson,B.W.,Redell,D.D., Experience with Process and Monitors in Mesa, C-ACM, vol. 23, no .2, 1980.

6. Brinch Hansen,P., Concurrent Programming Concepts, Computing Surveys, vol.5, no. 4, 1973.

7. Popek,G.J. et al., Notes on the Design of Euclid, SIGPLAN Notices 12, 3, 1977.

8. Lampson, B.W. et al., Revised Raport on the Programming Language Euclid, 1978.

9. Wulf,W.A. (ed.), An Informal Definition of Alphard, CMU-CS-78-105, Dept. of Computer Science, Carnegie-Mellon University, 1978.

10. Reference Manual for the Ada Programming Language, Proposed Standard Document, U. S. Department of Defense, 1980.