프로그래밍 言語에서 aliasing에 관한 小考

金 炳 喆

A Review of Aliasing in Programming Languages

Kim Byung-chul

Summary

Goto statements, side effects, and aliasings etc are harmful features in programming languages. The definition of an aliasing is stated. By exemplifying some cases it is investigated that why it does occurr. Further the influences of aliasings are examined.

And how to resolve the aliasing problem is also discussed in some respects.

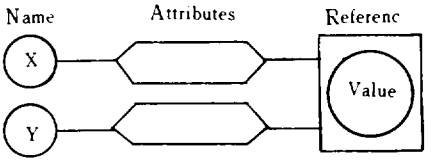
序 論

프로그래밍 言語의 유해한 특징으로는 go to문, 부작용, aliasing 등을 들 수 있다. 이들은 프로그램의 正當性 檢證, 프로그램 명세 등을 어렵게 하며 프로그램의 理解를 힘들게 한다. 특히 부작용과 aliasing은 최적화 등 컴파일링에도 障害가 된다.

자료 대상은 生存期間 중 한가지 이상의 명칭을 가질 수 있다. 즉 그 자료 대상에 대하여 相異한 명칭을 提供하는 相異한 참조환경을 여러가지로 관련시킬 수 있다. 예를 들면 資料對象을 call-by-reference에 의하여 부프로그램의 매개변수에게 移送하면 그것은 부프로그램에서는 형식인수로 참조 될 수 있고, 호출프로그램에서는 원래의 명칭을 有持할 수도 있다. 이와 반대로 하나의 資料對象은 pointer linkage를 통하여 여러가지 자료 대상의 한 성분으로 될 수도 있어서 그것에 접근할 수 있는 여러가지 합성명을 가질 수도 있다. 같은 자료 대상에 대한 복수 개의 名稱은 거의 모든 프로그래밍 言語에서 여러가지 方法으로 가능하다.(Pratt,1984).

定 義

어떤 자료 대상이 단위 참조환경에서 한개이상의(단순 또는 합성) 名稱으로 보여질 수 있을때 各 名稱은 그 자료 대상의 *alias*라고 한다. 다시말하면 같은 단위 Activation에서 한개 이상의 名稱으로 같은 자료 대상에 接近할 수 있는 可能性을 *aliasing*이라고 한다.



위 그림에서 X와 Y는 모두 같은 storage location (즉 memory address)을 가리킨다. 이때 X, Y는 서로의 alias라고 한다. (Horowiz. 1984).

또한 변수가 다른 변수의 alias를 갖도록 허용하는 프로그래밍 言語는 aliasing을 허용한다고 한다. 대부분의 프로그래밍 言語는 이를 許容하며 FORTRAN의 EQUIVALENCE문처럼 어떤 言語에서는 이것을 가능케하는 미캐니즘을 提供하기도 한다. 그리고 한 변수의 값을 變換시키면 自動的으로 同一한 alias를 갖는 모든 변수들의 값이 변하는 상태를 *aliasing* 效果라고 한다.

發　　生

자료 대상이 복수 개의 名稱을 가질 때 그것이 나타나는 각 참조환경에서 唯一하다면 問題는 일어나지 않는다. 그러나 같은 참조환경에서 相異한 名稱을 사용한 同一한 자료 대상을 참조할 수 있다면 言語 사용자와 설계자 양방에서 重大한 問題를 야기시킨다.

```
Program MAIN (OUTPUT);
  Procedure SUB 1 (Var J: Integer);
    begin
      :                 } J는 볼 수 있으나 I는 볼 수 없다.
    end;
Procedure SUB 2
  Var I : integer
  begin
    :
    SUB (I);            } I는 볼 수 있으나 J는 볼 수 없다.
    :
  end;

  begin
    :
    SUB (2);            } I도 J도 볼 수 없다.
    :
  end.
```

(a) No aliasing

```
Program MAIN (OUTPUT);
  Var I : integer;
  Procedure SUB 1 (Var J : integer);
    begin
      :                 } I와 J는 같은 자료를 참조한다.
    end;

Procedure SUB 2
  begin
    :
    SUB 1 (I)           } I는 볼 수 있으나 J는 볼 수 없다.
  end;
  begin
    :
    SUB 2               } I는 볼 수 있으나 J는 볼 수 없다.
  end
```

(b) I와 J가 SUB 1에서 aliases.

위 두 Pascal 프로그램에서 정수형 변수가 프로그램의 실행중에 다른 지점에서 두 名稱 I와 J를 갖는다. (a)의 경우 같은 프로그램에서 사용되는 두 名稱 I와 J가 실행 중 어떤 지점에서도 사용될 수 없기 때문에 aliasing이 일어나지 않는다. (b)의경우 부프로그램 SUB 1에서 I와 J는 같은 자료 대상에 대한 alias이다. 왜냐하면 I가 by-reference로 SUB 1에 pass되고 그 곳에서 名稱 J와 관련이 되고 동시에 I는 SUB 1에서 비지역 변수로 가시적이기 때문이다. (Pratt. 1984).

事　　例

aliasing이 일어나는 경우는 크게 다음 두 가지로 분류할 수 있다.

1) 동일 단위 프로그램 內에서 aliasing이 可能하도록 하는 미캐니즘을 提供하는 경우

(예) FORTRAN에서

```
EQUIVALENCE (A, B)
A=5.4
B=5.7
```

Write (6, 10) A라고 하면 A의 값은 5.7이 出力된다

2) 매개변수에의 移送이 있는 경우. (Pratt. 1984).

(1) 비지역 변수와 형식 매개 변수간의 aliasing로 실인수인 대상을 call-by-reference로 移送하고 부프로그램에서 비지역 변수도 同一한 대상을 참조

하면 aliasing이 일어난다.

(예) Pascal에서

```
Praocedure SWAP 1 (Var X:integer);
  begin X:=X+a;
        X:=X-a;
        X:=X-a

  end;
```

(2) 두 형식 매개변수의 aliasing은 같은 대상이 실인수의 값으로 call-by-reference에 의하여 인용된 경우에 일어난다.

(예) Pascal에서

```
Procedure SWAP 2 (Var X, Y: integer);
  begin X:=X+Y;
        Y:=X-Y;
        X:=X-Y
  end;
```

(1)에서 SWAP 1 (b), b≠a (2)에서 SWAP 2 (a, b), a≠b이면 aliasing은 일어나지만 問題는 發生되지 않는다. 그러나 이때 SWAP1(a)나 SWAP2(a, a)로 호출하면 반환되는 a의 값은 항상 0이 된다. 따라서 aliasing이 일어나는 경우라 할지라도 두 자료대상이 중첩(overlapping) 되는 경우에만 誤謬가 發生한다.

2)의 경우에 Call-by-reference에 의하지 않고 다른 기법으로 인수를 傳達하면 aliasing은 일어나지 않는다. 따라서 例를 들어 Call-by-value -result에 의한 경우에는 SWAP1(a), SWAP2(a, b)는 aliasing 조차 일어나지 않으므로 誤謬가 없다. 또 aliasing에 관한 問題는 단순변수 뿐만 아니라 배열요소인 첨자변수나 Pointer형 변수에 대해서도 그대로 適用된다. 그리고 ALGOL 68에서는 named stack object를 가리키는 것으로 aliasing이 일어난다. 또 pascal type의 시스템에서 가변레코드는 aliasing을 일으킨다. 이는 Ada에서 해결되었다. 그리고 LISP에서 sublist의 sharing은 問題가 되지 않는다.

그 이유는 리스트 처리함수는 순수함수이므로 의 리스트를 破壞하지 않을 뿐만 아니라 sublist의 sharing은 사용자에게 보이지 않기 때문이다(Mac-Lennan, 1983).

有 害 性

오늘날 aliasing은 信賴度에 부정적 영향을 미치는 것으로 간주되고 있다.

또 값의 變更은 프로그램 本文에 明示되어야 하며 aliasing을 통하여 음성적인 變更은 許用되지 않으려는 신념이 있다. 그래서 프로그래머, 독자, 구현자들로 하여금 때때로 다른 名稱이 같은 자료 대상을 가리키고 있어서 부프로그램을 理解하기 어렵게 만든다. 이 현상은 부프로그램의 조사로 發見될 수 없고 부프로그램을 호출하는 모든 단위프로그램을 조사해야 可能하다. aliasing의 結果로 부프로그램은 예기치 않은 그리고 잘못된 結果를 낳을 수도 있다.

(1) aliasing은 프로그램을 이해하기 어렵게 만든다.

例를 들면 다음의 일련의 두 문

X:=A+B;

Y:=C+D;에서 X와 Y에의 치환은 명백히 獨立的이고 위 순서대로 이거나 만약 X가 차후에 참조되지 않는다면 처음 치환문은 삭제될 수도 있다. 그러나 X와 C가 alias라고 하면 문들은 사실 相互依存的이고 誤謬를 유발하지 않고서는 순서를 바꾸거나 삭제할 수 없다.

(2) aliasing은 프로그램의 正當性을 확인하기 어렵게 만든다.

왜냐하면 어느 두 변수명도 반드시 상이한 자료대상을 참조하는 것으로 가정되어 있지 않기 때문이다. 가끔 분리된 解釋이 aliasing이 나타나는 가를 결정하는 데 필요하다. 프로그램의 正當性을 證明하는 일은 言語 설계에서 aliasing이 탐탁치 못함을 보이는 도구로 되어 왔다.

(3) aliasing은 다음과 같이 변역 중에 프로그램 코드의 최적화를 防害하고 있다. 예를 들면,

a:=(X-Y*Z)+W;

b:=(X-Y*Z)+u;에서 만약 a가 X,

Y 또는 Z의 alias이면 부분식 X-Y*Z가 한번만으로 평가될 수 없다.

對　　策

같은 자료 대상에의 여러가지 接近 경로의 存在는 어떤 의미론적 제약을 확인하고 어떤 최적화의 摘用可能性을 조사하는 데 필요한 data flaw analysis(치환의 解釋과 패턴 사용)를 복잡하게 만든다. 例를 들어 컴파일러 製作者들이 치환을 연기하고자 하면 새로운 값에의 接近을 다른 接近 경로상에서 시도 되지 않을 것으로 확신해야 한다. 이 複雜性을 *aliasing problem*이라고 한다. (Waite Goos. 1984).

계산 중에 step의 순서를 바꾸거나 불필요한 스텝을 삭제하는 일이 종종 필요하다. aliasing이 일어날 수 있는 곳에서는 명백히 獨立的인 두 계산스텝은 aliasing에 의존하지 않음을 보증하는 추가적인 解釋없이는 不可能하다.

aliasing으로 야기된 問題 때문에 때때로 새로운 言語 設計는 alias를 許用하는 특징을 制限하거나 除去하려고 시도한다. 그래서 프로그래밍 言語 GYPSY와 Euclid는 1970년대 후반에 프로그래밍 言語의 유해한 특성을 除去하고자 Pascal에 기초하여 設計된 言語이다. GYPSY는 전역변수를 提供하지 않으며, 특히 aliasing 제거에 중점을 둔 Euclid는 전역변수는 提供하지만 이들의 接近을 제어하는 매케니즘을 가지고 있다.

Call-by-value-result의 기법에도 실인수의 주소는 입구와 출구에서 두번 計算되므로 특히 실인수가 배열 요소이면 부작용에 의하여 첨자식의 變更될 可能性이 있으므로 問題가 있다. 또한 하나 이상의 매개변수가 한 값을 나르면 재 복사의 순서가 정의되지 않아 다른 結果를 만들 수도 있다. 이 해결책은 매개변수 計算이나 移送에 대한 뛰어난 계책에 있지 않고 aliasing의 可能性을 불허하는데 있다. (Horowitz. 1984).

aliasing을 除去하는 기본적인 두 가지 方法은 첫째로 aliasing을 일으킬 수 있는 특징(例를 들면 Pointer, 참조 매개변수, 전역변수, 배열 등)을 완전히 사용하지 않는 것으로 이는 言語를 정말 허약하게 만든다.

둘째로는 Eucid에서와 같이 aliasing의 可能性을 지배하는 특징의 사용에 제한을 가하는 것이다.

(1) 실인수의 overlapping을 막기 위하여 상이한 형식인수는 상이한 실인수를 넘겨 받도록 制限하는 바 이것은 자칫 言語의 사용을 서투르고 어렵게 만든다. 그래서 Euclid에서는 array의 index와 같이 Collection variable을 도입하여 pointer에 대해서도 상이한 條件인 legality assertional을 번역기에 의하여 生成토록 規定하여 testing 단계에서 이것이 自動的으로 실행기간 체크로 컴파일되게 한다. 그리하여 실행시간에 이것이 거짓으로 되면 실행을 中斷하고 메시지를 내어 보낸다. 이로서 Call-by-reference가 마치 Call-by-copy와 같은 역할을 한다.

(2) 전역변수의 變更을 明示的으로 規定하는것이다.(Ghezzi와 Zezayeri, 1982) 전술한 바와 같이aliasing은 전역변수와 절차의 형식 매개변수 간에도 일어난다. Euclid에서는 전역변수가 필요시 되면 明示的으로 부프로그램에 의하여 import되어야 한다. 그래서 부프로그램이 호출되는 모든 scope에서 접근 가능해야 한다. import된 각 변수에 대해서 그것이 읽혀질 것인가, 쓰여질 것인가, 또는 둘 다인가를 지시할 필요가 있다.

이리하여 수정 可能한 전역변수는 참조에 의하여 傳達된 暗示的으로 부가될 매개변수처럼 aliasing탐지 알고리즘에 의하여 取扱될 수 있다. 전역변수의 明示的 importation은 프로그래머로 하여금 한 節次內의 변수의 集合을 외부 scope에서 선언된 (mask 되지 않은) 변수들의 어떠한 부분집합에서도 가시적이도록 制限을 허용한다. 이리하여 번역기는 한 단위 內에서 오직 가시적인 변수들만 接近될 수 있고 어떠한 接近도 합법적이라는 것-例를 들면, 오직 읽기만 하는 변수는 수정될 수 없다-을 保證할 수 있어야 한다. 이것은 순수 ALGOL 같은 Scope 상에서 유리하다. 특히 대형 프로그램의 경우에 그 내부 절차는 포위하는 Scope에서 선언된 모든(mask 되지 않은) 변수들을 自動的으로 유전한다. 또 이는 그 변수들을 어떤 제어되지 않은 方法으로 수정할 수 있다.

Euclid함수는 by-reference매개변수를 갖도록 許用되지 않고 단지 읽기변수만 import할 수 있다. 이

리하여 그들 실행을 부작용을 일으키지 않고 수학적 함수처럼 動作한다. 절차에서 aliasing을 許用하지 않은 중요한 結果는 참조에 의한 매개변수 傳達이 복사에 의한 傳達과 등치이다. 그런고로 매개변수 傳達을 어떻게 구현하는가의 選擇은 排他的으로 效率性 이유에 根據한 번역기에 의하여 이루어질 수 있다. Ada는 매개변수 傳達이 참조에 의하여 구현되어야 하는가 또는 복사에 의하여 구현되어야 하는가를 規定하지 않는다. 그러나 Euclid와는 달리 비합법적, aliasing을 갖는 프로그램은 프로그램 검증기에 의하여 실행시간 전에 잡히도록 요구하지 않는다. 이래서 몇몇 비합법적 Ada 프로그램은 잡히지 않는 채로 남아있게 될 수 있으며, 結果的으로 같은 프로그램에 대하여 상이한 매개변수 傳達 구현은 상이한 結果를 낳을 수 있다.

Euclid의 해결책이 흥미롭고 깨끗하지만 Euclid에 의하여 부가될 몇 가지 제약을 傳統的 컴파일러에 강요할 수 없으며 프로그램 검증기를 포함하는 프로그램 開發시스템을 요구할 수 없다. 특히 모든 legality assertion은 검증기에 의하여 證明될 필요가 있다. Euclid식 接近의 성공은 실제 專門家들에 의한 프로그램 檢證의 성공에 거의 달려 있다. 현재 실제 응용에서 대부분이 프로그램 檢證의 경험이 적기 때문에 이것은 논쟁의 이슈가 되고 있다.

한편 많은 사람들은 aliasing은 해로운 것이며 또 매우 유용한 특징으로 생각하고 있는 점도 주목해야 할 것이다.

摘　　要

aliasing은 program을 理解하기 어렵고, program의 正當性을 確認하기 어렵게 만들며, program 번역 중에 program 코드의 최적화를 방해하고 있는 것으로 理解되고 있으나, 이러한 有害性에 대한 對策은 aliasing을 일으킬 수 있는 특징을 使用하지 않는 것으로, 이것은 言語를 허약하게 만들게 됨으로 Euclid에서와 같이 aliasing의 可能性을 지배하는 특징의 使用에 제한을 가하여 Call-by-reference가 마치 Call-by-copy와 같은 역할을 하게 하고, 全域 變數의 變更을 明示的으로 規定하는 것이다.

參 考 文 獻

MacLennan. B. J. 1983. Principles of Programming Languages, CBS Colledge Publishing, p. 378.

Ghezzi, C. Medhi. J., 1982. Programming Languages Concepts. John Wiley & Sons, Inc., pp. 183-188.

Horowitz, E. 1984. Fundamentals of Programming Languages. Computer Science Press, 2nd ed., pp. 79, 213-214.

Pratt. T. W. 1984. programming languages (2nd ed.). Prentice-Hall, Inc., pp. 221-222, 264-265.

Waite. W. M. Gerhard Goos, 1984. Compiler Construction. Springer-Verlag, New York Inc., pp. 34, 329.