이해와 구현이 용이한 C 언어 시퀀스 포인트 모델

전웅[O] 한동수
한국정보통신대학교
{woong[O], dshan}@icu.ac.kr

Comprehensive and Algorithmical Sequence Point Model for C

Woong Jun[0] Dong-soo Han
Information and Communications University

요 약

이식성과 성능 사이의 타협을 위해 수식 평가 순서를 부분적으로 정의할 수 밖에 없는 C 언어는 표준이 사용하는 일반 영어의 모호성으로 일부 복잡한 수식의 유효성(validity)을 판별하기 어려운 문제를 안고 있다. 그동안 의존 관계나 언어 형식화(formalization)를 이용해 일관되게 유효성을 판별하려는 시도가 있었으나 이해와 구현 모두가 용이해야 한다는 표준의 요구를 만족하지 못했다. 본 연구에서는 구현을 고려해 AST (Abstract Syntax Tree)에 변수의 참조·수정 정보를 덧붙여 수식 부작용(side effect)과 관련된 시퀀스 포인트 (sequence point) 문제를 해결하는 효율적인 모델을 제안한다.

1. 서 론

C 언어는 언어 정의를 일정 수준으로 규제해 고수준의 이식성을 제공하면서 동시에 기반 실행 환경의 특성을 최대한 활용해 높은 성능을 끌어낼 수 있도록 설계된 고급 언어로, 언어 사용자와 언어 구현자(implementer) 사이의 조율을 담당하는 주체는 ISO/IEC에 의해 제정된 언어 표준[1, 2]이다. 표준은 접근성(accessibility)을 위해 언어를 형식화하는 대신 BNF와 일반 영어를 사용해 문법과 의미를 정의하기 때문에 일부 경우에 표준이 제공하는 정의가 불분명한 경우가 발생하고 있다[3]. 일부 복잡한 형태를 갖는 수식의 유효성(validity) 판별 역시 그와 같은 경우에 해당한다.

명령형 언어(imperative programming language)인 C 언어로 작성된 프로그램은 본질적으로 실행 과정에서 변수값을 변화시키는 행동(action)을 명시하는 수식으로 구성된다. 하지만, 수식 평가에 일정 순서를 엄격하게 강요하는 경우 성능 저하가 불가피하기 때문에 표준은 문법적으로 정의되는 우선순위(precedence)와 결합방향(association)으로 수식의 부분적인 평가 순서만을 규정하고, 그 외에 피연산자 평가 순서와 부작용(side effect) 발생 순서에 대해서는 정의하지 않는다. 특히, 수식 내 피연산자나 부분 수식(sub-expression)의 평가가 부작용의 발생을 수반하지 않아야 한다.

단, 프로그램 내 일정한 지점에서 수식의 부작용이 완료되도록 하기 위해 이전 수식이 포함하는 부작용이 완료되고 다음 수식이 포함하는 부작용이 시작되지 않는 지점인 시퀀스 포인트(sequence point)를 적절한 위치에 배치하고 있다. 또한, 시퀀스 포인트를 통해 유효한 수식과 그렇지 않은 수식을 구분하기 위한 규칙(두 시퀀스 포인트 사이에서 한 변수는 최대 한번만 수정될 수 있으며, 수정되는 변수의 값은 저장될 새 값을 결정하기 위해서만 사용되어야 한다)을 제공하고 있지만, 영어 표현의 모호성으로 실제 프로그램에서 사용되는 일부 수식의 유효성을 올바르게 규정짓지 못하는 문제를 안고 있다.

시퀀스 포인트 모델(sequence point model)은 수식에 강제되는 부분적인 순서만을 바탕으로 수식의 유효성을 명확하게 결정할 수 있는 규칙의 집합을 말한다. 보다 명확하게 정의된 규칙을 통해 컴파일러나 정적 분석 도구가 잠재적으로 이식성과 관련된 문제 혹은 직접적인 오류가 되는 부분을 사전에 지적할 수 있도록 하는 것이 가능하다.

초기 C 언어 수식의 유효성 판별에 대한 연구는 대부분 시퀀스 포인트에 대한 이해 부족과 명확한 규칙 부재로 기본적인 형태의 수식도 올바르게 판별하지 못했다[4]. 시퀀스 포인트 문제에 대한 인식으로 1999년표준 개정 과정에서 수식의 가능한 평가 순서를 바탕으로 의존 관계를 파악하는 접근법을 시퀀스 포인트 모델로 제시했으나 모델이 갖는 문제로 최종 표준안에서는 배제되었다[5]. 또한, C 언어의 언어 정의 일부를 형식화하기 위한 접근으로 HOL (high-order logic)을 통해 C 언어의 데이터형 체계(type system)와 수식 체계를 형식화하는 시도도 있었다[6, 7]. 하지만 두 방법 모두 표준이 요구하는 중요한 두 특징인 이해의 용이성(특정 기술에 대한 전문 지식 없이도 이해가 가능해야 함)과 구현의 용이성(알고리즘의 형태를 갖추되 기존의 컴파일러나 분석 도구에 어렵지 않게 수용될 수 있어야 함)을 부분적으로 만족하지 못하고 있다.

본 연구에서는 수식의 AST(Abstract Syntax Tree) 표현을 바탕으로 일정 규칙을 기계적으로 따라 수식의 유효성을 판단할 수 있는 효율적인 접근법을 제안한다.

2. 꼬리표 AST(tagged AST)를 이용한 시퀀스 포인트 모델

2.1 가정

모델 제안에 앞서 문제를 간략화하고 중심이 되는 시퀀스 포인트 문제에 집중할 수 있도록 몇 가지 가정을 명시한다.

> 가정 1. 수식을 분석하는 과정에서 피연산자 및 연산 결과의 데이터형(type)은 고려하지 않는다. 데이터형 역시 수식의 유효성에 영향을 주지만 이는 시퀀스 포인트 모델과는

무관한 데이터형 체계에 의한 것이다. 동시에 수식 내에서 데이터형과 관련되어 쓰이는 sizeof 연산자와 캐스트 연산자 역시 무시하기로 한다.

가정 2. 에일리어싱(aliasing) 문제는 프로그램 정적 분석의 본질적인 한계를 드러내며 문제를 복잡화하기 때문에 고려하지 않기로 한다. 따라서 분석하는 수식 내에서 서로 다른 변수는 항상 서로 다른 메모리 공간을 점유하는 것으로 가정한다.

가정 3. 데이터형을 배제하더라도 분석 대상의 수식은 표준 컴파일러에 의해 올바르게 번역될 수 있는 수식으로 제한한다. 이는 피연산자가 일부 생략된 불완전한 수식 역시 배제한다.

2.2 연산자 분류

C 언어는 설계 당시의 타 언어와 비교해 상대적으로 다양한 연산자를 제공한다. 하지만, 시퀀스 포인트 모델을 다룰 때 특정 연산의 의미(semantic)는 중요하지 않기 때문에 각 연산자별로 규칙을 열거하는 것은 무의미하다. 그보다는 시퀀스 포인트 모델과 관련된 특성을 추려 공통된 특성을 대표하는 연산자로 묶는 것이 경제적이다. 따라서 본 연구에서는 다음과 같은 대표 연산자만을 다룬다.

표 1. 대표 연산자 분류

+	두 피연산자에 연산을 취해 결과를 내는 이항 연산자(* / % - << >> < > <= >= == != & ^ |)를 대표한다.
=	좌측 피연산자의 값을 우측 피연산자로 주어진 좌변값(lvalue)에 저장하는 연산자를 대표한다.
+=	두 피연산자에 연산을 취해 결과를 좌측 피연산자로 주어진 좌변값에 저장하는 이항 연산자(-= *= /= %= <<= >>= &= ^= |=)를 대표한다.
,	이례적으로 좌측 피연산자와 우측 피연산자 사이의 평가 순서를 정의해주며, 두 피연산자의 평가 사이에 시퀀스 포인트를 정의해주는 이항 연산자(&& ||)를 대표한다.
[]	포인터 간접 지정 연산자 *를 수용할 수 있는 배열 첨자 지정 연산자를 대표한다.
.	[]와 함께 사용되어 구조체 및 공용체 포인터 멤버 지정 연산자인 ->를 수용할 수 있는 구조체 및 공용체 멤버 지정 연산자를 대표한다.
++	피연산자에 연산을 적용해 그 결과를 다시 피연산자로 주어진 좌변값에 저장하는 단항 연산자(--)를 대표한다.

대표 연산자 중 '+' '=+' ', ' '[]'는 연산 결과를 생성하기 위해 양쪽 피연산자를 모두 필요로 하며, '='는 우측 피연산자만을 필요로 한다. 또한, '='와 '=+'는 좌측 피연산자의 값을 수정하며, '++' 역시 피연산자의 값을 수정한다. 이와 같은 특성을 바탕으로 수식의 유효성을 결정하는 알고리즘은 다음과 같다.

2.3 수식 유효성 결정 규칙

수식의 유효성을 결정하기 위해 AST를 기본으로 AST의 연산자 노드에 덧붙을 수 있는 수정 테이블(modification table)과 참조 테이블(reference table)을 도입한다. 이 두 테이블은 각각 변수를 쓰고, 읽는 과정을 추적하는데 사용된다.

알고리즘은 크게 두 단계로 구성되어 있다. 첫 번째 단계에서는 수식 AST의 연산자 노드에 적절한 테이블을 덧붙이는 과정이며, 두 번째 단계는 연산자 노드의 테이블을 상위 노드로 통합하면서 실제 수식의 유효성을 판별하는 과정이다. 우선 첫 번째 단계의 알고리즘은 다음과 같다.

1. 주어진 수식을 표현하는 AST를 구성
2. 최상위 노드를 시작으로 BFS(Breadth-First Search)를 수행하며 다음과 같은 과정을 수행
   2-1. 노드의 연산자가 '++'인 경우,
      2-1-1. 자식 노드가 연산자가 아니면 수정 테이블에 피연산자를 추가
      2-1-2. 자식 노드가 연산자이면 전체 규칙을 재귀적으로 적용 (재귀 적용 후 수정 테이블에 추가)
   2-2. 노드의 연산자가 '++'가 아닌 경우,
      2-2-1. 좌측 자식 노드가 연산자가 아닌 경우,
         2-2-1-1. 현재 노드의 연산자 종류에 따라 다음과 같은 행동을 취함
            +, [] . 피연산자를 참조 테이블에 추가
            = =+ 피연산자를 수정 테이블에 추가
      2-2-2. 좌측 자식 노드가 연산자이면 전체 규칙을 재귀적으로 적용 (재귀 적용 후 규칙 2-2-1-1에 맞춰 테이블에 추가)
      2-2-3. 우측 자식 노드가 연산자가 아닌 경우,
         2-2-3-1. 현재 노드의 연산자 종류에 따라 다음과 같은 행동을 취함
            + = =+ , [] 피연산자를 참조 테이블에 추가
            . 우측 노드는 멤버명이기에 무의미함
      2-2-4. 우측 자식 노드가 연산자이면 전체 규칙을 재귀적으로 적용 (재귀 적용 후 규칙 2-2-3-1에 맞춰 테이블에 추가)

이 알고리즘을 수행하는 과정에서 피연산자가 상수로 주어진 경우는 유효한 행위가 필요 없으므로 무시될 수 있다.

두 번째 과정은 BFS(Breath-First Search)의 역순으로, 하위 노드부터 최상위 노드로 각 연산자 노드의 수정 및 참조 테이블을 상위 연산자 노드로 전달하면서 다음과 같은 규칙으로 검사를 수행한다. 여기서는 편의상 특정 피연산자가 참조 테이블에 추가되는 경우를 "참조가 일어나는 경우"로, 수정 테이블에 추가되는 경우를 "수정이 일어나는 경우"로 표현한다.

1. 피연산자에 참조가 일어나는 경우 참조 테이블에 참조되는 노드의 위치를 기록한다.
2. 피연산자에 수정이 일어나는 경우,
   2-1. 단일 피연산자에 수정이 두 번 일어나는 경우,
      2-1-1. 노드의 연산자가 ','인 경우에는 상위 연산자 노드로 한 번의 수정만을 전달
      2-1-2. 그렇지 않은 경우 수식은 정의되지 않음
   2-2. 참조된 단일 피연산자에 수정이 일어나는 경우,
      2-2-1. 노드의 연산자가 ','인 경우에는 상위 연산자 노드로 한 번의 수정만을 전달
      2-2-2. 피연산자의 참조 중 하나라도 수정이 이루어진 연산자의 (재귀적인) 자식 노드가 아닌 노드에서 이루어진 경우, 수식은 정의되지 않음
      2-2-3. 피연산자의 모든 참조가 수정이 이루어진 연산자의 (재귀적인) 자식 노드에서 이루어진 경우,
         2-2-3-1. 참조가 일어나는 모든 노드가 '=' 연산자의 (재귀적인) 좌측 자식 노드가 아니라면 상위 연산자 노드로 한 번의 수정만을 전달
         2-2-3-2. 하나의 참조라도 '=' 연산자의 (재귀적 인) 좌측 자식 노드라면 수식은 정의되지 않음

이때 알고리즘에서 ',' 연산자를 별도로 다루는 이유는 다른 연산자는 가지고 있지 않은 시퀀스 포인트를 포함하기 때문이며, 규칙 2-2-3-1이 존재하는 이유는 단순 대입 연산자 '='의 경우 연산의 결과를 결정할 때 좌측 피연산자는 무관하기 때문이다.

2.4 규칙 적용 결과

지금까지 정의한 모델을 실제 몇 가지 전형적인 예에 적용한 결과는 다음과 같다.

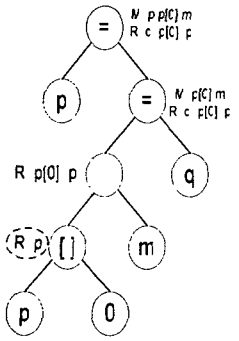

그림 1. p = p->m = q;

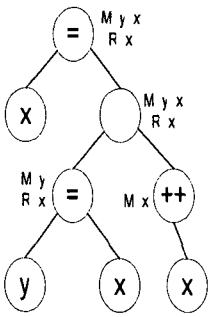

그림 2. x=(y=x),x++;

각 노드 옆에는 설명한 두 단계의 과정이 모두 진행된 후 수정(M) 및 참조(R) 테이블의 결과를 간략하게 보여주고 있다.

그림 1에 주어진 수식은 최상위 노드에서 두 번째 단계의 규칙 2-2-3-2가 적용되어 정의되지 않은 수식으로 판별된다. 이는 두 시퀀스 포인트 사이에서 수정되는 변수의 이전 값이 저장될 새 값 이외에도 값이 저장될 위치를 결정하는데 사용되는 경우를 보여준다.

그림 2에서는 , 연산자 노드에서 두 번째 단계의 규칙 2-2-1이 적용되지만, 최상위 노드에서 규칙 2-2-3-2가 적용되어 정의되지 않은 수식으로 판별된다. 이는 제시된 모델이 시퀀스 포인트에 의해 보호되는 부분을 정확히 구분해주고 있음을 보여준다.

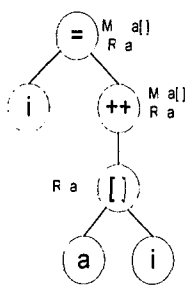

그림 3. i = ++a[i];

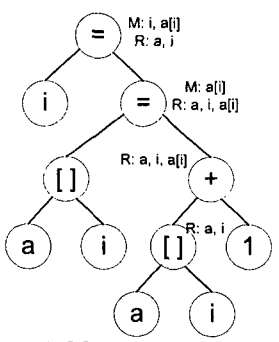

그림 4. i=(a[i]=a[i]+1);

그림 3에서는 최상위 노드에서 두 번째 단계의 규칙 2-2-3-1이 적용되어 올바른 수식으로 판별된다. 그림 4에서는 최상위 노드에서 i에 일어나는 두 번의 참조를 검사할 때 하나에 규칙 2-2-3-2가 적용되어 정의되지 않은 수식으로 판별된다. 그림 3과 4는 의미상으로는 (피연산자)=(피연산자)+1과 유사하지만, 피연잔자를 한번만 평가하는 '++' 연산자의 효과를 분명하게 보여주고 있다. 또한, 그와 같은 특성으로 '++' 연산자의 전위형/후위형의 구분이 수식 유효성 판별에는 영향을 주지 않음을 알 수 있다.

2.5 단항 및 조건 연산자와 함수 수식

3.2절에서 다루지 않은 조건 연산자 '?:'의 경우 첫 번째 피연산자의 평가 결과에 따라 두 번째 혹은 세 번째 피연산자만을 평가하고, 첫 번째와 두 번째 혹은 첫 번째와 세 번째 피연산자 사이에 시퀀스 포인트가 정의되기 때문에 경우에 따라 다음 두 수식 중 하나로 완전히 대체될 수 있어,

(첫 번째 피연산자) && (두 번째 피연산자)
!(첫 번째 피연산자) && (두 번째 피연산자)

결국 시퀀스 포인트 모델에서는 ',' 연산자로 대체가 가능하다. 유사하게, '++', '--'가 아닌 단항 연산자의 경우 항상 피연산자를 참조만 하기 때문에 따로 대표 연산자를 지정할 필요가 없다.

C 언어에서 함수 호출은 호출 직전과 직후에 시퀀스 포인트가 존재한다. 따라서 함수가 실행되는 의미적 영향을 무시하고 일종의 연산자로 취급할 경우 결국 하나의 시퀀스 포인트를 포함하는 ',' 연산자와 다르지 않다. 단, 함수 호출에 주어진 인자들은 연산자가 아닌 구두점(punctuation) ','로 구분되기 때문에 인자 평가 과정에는 시퀀스 포인트가 개입하지 않는다.

3. 결론 및 향후 연구과제

본 연구를 통해 필수적인 정보가 첨가된 노드를 갖는 AST를 간단하고 체계적인 규칙으로 분석함으로써 수식의 유효성을 모호함 없이 결정할 수 있음을 확인하였다. 본문에서 제시된 알고리즘은 사실상 트리 순회(tree traversal)로서 구현이 용이하며, 더구나 두 번째 단계는 트리 순회 과정에서 재귀 호출을 거슬러 돌아오는 과정에 해당하기에 수식 내 연산자 개수에 비례하는 수행 시간으로 유효성을 판별할 수 있다.

그러나 정의되지 않음(undefined)/정의됨(defined) 이외에 수식의 추가적인 상태인 명시되지 않음(unspecified)을 구분하기 위해서는 호출되는 함수 내 수식 역시 유사한 과정의 분석이 필수적이며, 또한 데이터형의 상등 관계를 구분할 수 있는 소규모 데이터형 체계를 도입함으로써 a[a[i]]=1;과 같은 수식에서 에일리어싱을 통해 발생할 수 있는 잠재적인 문제 가능성을 추적해 정적 분석의 한계를 어느 정도 극복할 수 있을 것으로 기대된다.

참 고 문 헌

[1] ISO, *ISO/IEC 9899:1990: Programming languages - C*, ISO, Geneva, Switzerland, 1990

[2] ISO, *ISO/IEC 9899:1999: Programming languages - C*, ISO, Geneva, Switzerland, 1999

[3] ANSI, *Rationale for ANSI X3.159-1989*, ANSI, New York, 1990

[4] Gurevich, Y., & Huggins, J.K., *The semantics of the C programming language*, Börger, E., et al. (eds), Selected papers from csl'92, Lecture Notes in Computer Science, vol. 702, pages 274-308, 1993

[5] ISO, *ISO/IEC 9899:1999 FCD (JTC1/SC22 N2794)*, ISO, Geneva, Switzerland, 1999

[6] Michael Norrish, *C formalised in HOL*, PhD thesis, University of Cambridge, 1998

[7] Michael Norrish, *Deterministic expressions in C*, In 8th European Symposium on Programming, volume 1576 of Lecture Notes in Computer Science, pages 147-161, 1999