추상 구문 트리에서 시멘틱 트리로의 변환

손윤식[O], 고석훈, 오세만
동국대학교 컴퓨터공학과
{sonbug[O], shko, smoh }@dongguk.edu

Transformation of AST to Semantic Tree

Yunsik Son[O], Serkhun Ko, Seman Oh
Dept. of Computer Engineering, Dongguk University

요 약

의미 분석이란 프로그램의 각 구성요소의 결합이 의미적으로 타당한가를 분석하는 과정으로 최근 컴파일러의 제작에서 필수 불가결한 요소이며, 속성문법(attribute grammar)이나 경험적인 방법(manual method)으로 해결한다. 그러나 이러한 방법론들은 효율성이나 자동화 측면에서 제약성을 가진다.

본 연구에서는 이러한 단점을 보완하기 위해 의미 분석정보가 포함된 시멘틱 트리를 정의하고, 대부분의 컴파일러에서 사용되는 구문분석 결과물인 추상 구문 트리를 시멘틱 트리로 변환하는 기법을 제안한다. 시멘틱 트리 변환기법은 의미 분석과정을 시멘틱 노드 단위로 처리하므로, 의미 분석 과정이 일관적으로 수행되며 효율적이고, 타 자료구조로의 변환이 용이하며 자동화가 용이하다.

1. 서론

의미분석이란 프로그램이 의미적으로 타당한가를 분석하는 과정으로 컴파일러의 구성단계에서 전단부에 해당하는 부분이다. 의미분석은 일반적으로 속성문법이나 매뉴얼 메소드의 두 가지 방법론으로 분석을 하는데 속성문법은 의미 분석을 일관되게 처리할 수 있는 반면, 효율성이 떨어지고 문법이 변경될 경우 문법과 같이 기술된 루틴의 변경이 동시에 이루어져야 하는 단점이 있으며, 매뉴얼 메소드는 의미 분석을 일관적으로 처리하기 힘든 단점이 있다[1].

본 논문에서는 의미 분석을 위한 자료구조인 시멘틱 트리를 정의하고 중간언어인 추상 구문 트리(Abstract Syntax Tree)에서 시멘틱 트리로의 변환을 통하여 의미분석을 하는 기법을 제안한다.

시멘틱 트리는 프로그램 구문구조 정보와 의미 정보가 동시에 반영된 자료구조로써 효율적인 의미분석과 코드생성이 용이한 형태이며, 의미분석과정의 결과물이다. 트리 변환 메소드는 AST를 시멘틱 트리로 변환하며, AST의 노드에 해당하는 의미분석을 일관적으로 처리한다.

제안된 기법은 구문분석이 완료된 AST상에서 노드별로 트리 변환 메소드를 적용하여 의미분석과정이 이루어지므로, 속성문법 보다 효율적이고 일관되게 의미분석을 할 수 있는 장점이 있다.

2. 관련 연구

2.1 중간언어

중간언어는 컴파일러에 대한 연구가 진행되면서 발전한 개념으로, 컴파일러를 구성하는 각 모듈 사이를 연결해주는 역할을 한다. 중간언어는 컴파일러의 특징에 따라 설계되며, Polish 표기법, 3-주소 코드, 트리 구조 코드, 가상 기계 코드 등 다양한 형태가 존재한다.

최근의 컴파일러의 구성단계에서는 중간언어로 대부분 AST가 사용되는데, AST는 트리 구조 형태를 가지며 프로그램의 의미를 보다 효율적으로 표현할 수 있다. 특히 구문분석과정에서 문법-지시적(syntax-directed)방법으로 쉽게 트리를 구성할 수 있으며, 의미적으로 불필요한 정보를 제거하여 프로그램의 구문구조가 간결하게 표현된다[2][3].

2.2 의미분석

의미분석은 컴파일 과정에서 구문분석된 프로그램이 의미적으로 올바른가를 검증하는 과정으로 형검사(type checking), 자료흐름분석(data flow analysis), 제어흐름분석(control flow analysis) 및 프로그래밍 언어별로 가지는 의미적인 특징에 대한 분석을 한다.

일반적으로 컴파일러 구성에서 의미분석을 하는 방법으로는 프로그래밍 언어의 문법에 속성을 기술하여 처리하는 속성 문

법과 의미 분석을 직접 인터프리팅을 통해 계산하는 매뉴얼 메소드의 두 가지 방법이 있다.

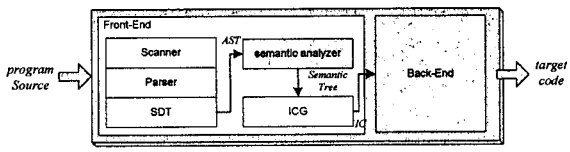
속성 문법을 이용하여 의미분석기를 구성하면 생성 규칙에 따라 일관되게 속성을 수집하고 분석이 가능하지만 심벌에 대한 속성을 처리하기 별도의 모듈(attribute evaluator)이 필요하며 속성 분석에 따른 복잡도가 증가한다. 또한 문법의 변경에 따른 의미 규칙 변경의 비용이 많고 복잡한 프로그래밍 언어의 구조를 분석하기가 어렵다.

매뉴얼 메소드는 속성 문법 등의 일반적인 방법론으로 해결이 어려운 의미 분석을 처리하기 위한 방법으로 심볼에 대한 속성과 자료의 흐름을 인터프리팅을 통해 분석한다. 매뉴얼 메소드는 일반적으로 트리 구조의 중간언어를 이용하여 의미 분석이 가능한 인터프리터를 구현하는데, 이러한 방법은 각각의 의미 분석만을 놓고 보면 효율적이지만 여러 가지의 의미 분석을 동시에 할 경우 효율성이 떨어지게 되고, 추가적인 의미 분석을 하기 위해서는 별도의 분석 모듈이 필요한 단점이 있다.

3. 트리 변환

3.1 시멘틱 트리

시멘틱 트리는 프로그램의 구문 구조 정보인 AST에 의미 정보가 반영된 이진 트리 형태의 자료구조이다. 시멘틱 트리는 프로그램의 구문 구조 정보를 효율적으로 표현하는 AST를 기반으로 정의하였기 때문에 AST가 가지는 특징을 모두 포함한다. 컴파일러의 구성 단계에서 시멘틱 트리의 위치는 다음과 같다[4].



[그림 1] 시멘틱 트리의 위치

시멘틱 트리는 프로그램의 의미정보를 구조적으로 표현이 가능하도록 설계되었으며, 그 기본 구성은 의미 정보 노드로 이루어진다. 각각의 심벌에 대한 의미 정보는 시멘티 트리의 의미 정보 노드가 관리하고 프로그램의 구조적인 표현은 의미 정보 노드로 구성된 트리의 형태와 AST의 프로그램 구조 노드의 결합으로 이루어진다.

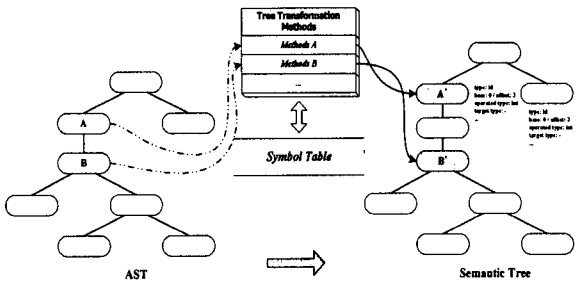
시멘틱 트리에서는 의미 분석 과정에서 사용하는 속성을 의미 정보 노드의 종류로 구분하여 각각의 역참조와 형 변환, 형 부가 연사자에 해당하는 노드를 구별한다.

의미 분석과정에서 속성 정보 수집 외에 필요한 또 하나의 작업은 코드 생성을 위해 추가적인 정보를 수집하는 작업으로 이러한 코드 생성을 위한 정보는 각각의 의미 정보 노드가 보유하게 된다. 이러한 의미 정보와 코드 생성 정보는 AST를 이용하여 의미 분석하는 과정에서 트리 변환 메소드에 의해 얻어진다.

시멘틱 트리를 설계하면서 의미 정보 및 구문 구주의 표현 외에 주안을 둔 또 하나의 부분은 트리 변환 메소드을 이용하여 의미 분석을 하기 위한 전체 조건으로 시멘틱 트리의 구조와 의미 정보 노드가 유일성을 가져야 한다는 부분이다. 시멘틱 트리의 구조와 의미 정보 노드는 각각의 프로그램 구조에 따라 유일성을 가지게 함으로써 일관되게 트리 변환 메소드를 적용시킬 수 있게 된다.

3.2 트리 변환 메소드

트리 변환 메소드는 AST를 시멘틱 트리로 변환하면서 의미 분석을 하고, 시멘틱 트리에 의미 분석 결과를 반영한다. [그림 2]는 트리 변환 모델의 전체적인 모습이다.



[그림 2] 트리 변환 모델

AST는 프로그램의 구문 구조 정보를 표현하고 각 심벌에 대한 정보는 심벌 테이블에 표현한다. 이러한 정보는 AST의 노드별로 정의되어 있는 트리 변환 메소드를 통하여 수집되고 의미 분석 과정을 거친 다음 시멘틱 트리로 변환된다.

의미 분석을 위한 트리 변환 메소드는 크게 심벌의 속성 정보를 계산하는 메소드와 타입변환 메소드, 노드 변환 메소드, 그리고 프로그램의 제어 흐름을 분석하는 제어 흐름 메소드로 구성되어 있다. 우선 심벌의 속성 정보를 계산하는 메소드는 다음과 같은 형식으로 이루어진다.

심벌에 대한 정보는 심벌 테이블에 저장이 되어 있으며, 계산된 속성 정보는 의미 정보 노드에 저장된다. 이는 의미 정보 노드의 분류를 구문 구조 표현을 위한 연산 위주로 분류한 데에 따른다. 저장되는 정보로는 일반적인 의미 분석에 사용되는 심볼 속성과 타입 정보, 그리고 코드 생성을 위한 base/offset 등이다.

다음으로 타입 변환 메소드는 타입 변환이 가지는 특성(transparency)을 이용하여 AST의 노드를 타입 변환 의미 정보 노드로 변환한다[5]. 연산에 따른 타입 변환은 synthesized attributes를 이용하여 해당 메소드를 적용하는데 자식 노드의 타입 속성을 이용하여 미리 정의된 변환 규칙에 따라 트리를 확장한다.

일반적인 노드 변환 메소드는 다음 세가지 규칙을 따른다. 첫째, AST의 기본 노드는 전부 시멘틱 트리의 타입부가 노드로 변환한다. 둘째, AST 복합 연산 노드의 경우는 의미 분석과 코드 생성의 용이를 위하여 시멘틱 트리의 기본 연산 노드로 확장한다. 셋째, AST의 상수 연산 노드에 대해서는 연산결과를 반영하여 상수 노드로 대치한다.

마지막으로 프로그램의 제어 흐름을 분석하기 위해서 프로그램 상의 가능한 분기 점을 추적하여 그 흐름을 트리에 기술한다. 제어 흐름을 분석하기 위해서는 우선 프로그래밍 언어별로 정의된 분기문에 해당하는 AST 노드를 선택하여 이를 기준으로 기본 블록(basic block)을 구분하고 각 분기에 해당하는 노드 사이를 연결함으로써 제어 흐름을 표현한다.

4. 구현

다음은 제안된 트리변환 기법을 이용하여, ANSI C 프로그래밍 언어로 기술된 프로그램을 실제 의미 분석한 결과이다. 총 122개의 AST 노드와 C언어의 의미정보를 기반으로 245개의 의미정보 노드를 정의하였다.

[표 1] 트리 변환 메소드

의미 정보 추가	type determine address calculation
연산 노드 변환	basic node manipulation complex node manipulation check member operation check index operation constant folding
제어 흐름 추가	check control flow analysis

[표 2] 의미 분석 결과

```
// while (t!=t->next)
Nonterminal: WHILE_ST
   Nonterminal: NEI / opType: 3
      Nonterminal: CVP_I / opType: 3
         Terminal ( Type:id / Value:t / opType:6
            / targetType:45 / qualifier:0
            / (b:1, o:8, w:4) / Tag:1 / Dim:0)
         Nonterminal: CVP_I / opType: 3
            Nonterminal:ADDP / opType:6 / targetType:45
               Terminal ( Type:id / Value:t / opType:6
                  / targetType:45 / qualifier:0
                  / (b:1, o:8, w:4) / Tag:1 / Dim:0)
               Terminal (Type:int / Value:4 / opType:6)
```

5. 결론 및 향후 연구

시멘틱 트리는 의미분석을 위해 정의된 자료구조이며, 트리 변환은 기존의 의미분석 방법을 변형한 의미 분석 방법이다. 제안된 의미 분석 기법은 AST 노드에 따라 트리변환 메소드를 적용하여 효율적으로 의미정보가 반영되고, 코드생성정보를 포함한 시멘틱 트리로 변환한다.

본 논문에서는 시멘틱 트리와 트리 변환 기법을 사용하여 의미 분석 기법을 설계하고 범용 프로그래밍 언어를 대상으로 구현하고 실험하였다. 실험 결과 각각의 의미 정보 노드는 속성 값을 반영하여 확장되었으며, 트리변환 메소드를 통하여 변환되고 의미 분석 과정이 완료되었다.

향후 본 연구에서 제안한 의미 분석 기법의 효율성을 위하여, 의미 정보를 효과적으로 유지할 수 있는 자료구조에 대한 연구가 필요하며, 그리고 AST 노드와 시멘틱 노드, 트리변환 메소드의 매핑을 통한 의미 분석기 자동화 도구에 대한 연구가 필요하다.

참고문헌

[1] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen. Modern Compiler Design, John Wiley & Sons, 2000.

[2] A. V. Aho, R. Sethi, J. D. Ullman. Compilers, Principles, Techniques, and Tools, Addison Wesley, 1988.

[3] 오세만, 컴파일러 입문, 개정판, 정익사, 2004.

[4] B. M. Brosgol, "TCOLAda and the Middle End of the PQCC Ada Compiler," Proceeding of the ACM-SIGPLAN symposium on Ada programming language, Vol. 15 No. 11, pp.101-112, 1980.

[5] J. C. Mithell, "Coercion and Type Interface," 11th ACM Symp. on Principles of Programming languages, pp.175-185, 1984.

[6] D. E. Knuth, "The Genesis of Attribute Grammars," ACM Proceedings of the international conference on Attribute grammars and their applications, pp.1-12, 1990.

[7] Mark S. Sherman, Martha S. Borkan, "A flexible semantic analyzer for Ada," Proceeding of the ACM-SIGPLAN symposium on Ada programming language, Vol. 15 No. 11, pp.62-71, 1980.