동적 의미 분석에 의한 점진 해석기 구축

The Building of Incremental Interpreter Using Analyzing of Dynamic Semantics

한 정 란* 최 성**
Junglan Han Sung Choi

요 약

소프트웨어의 생산성을 향상시키기 위해 소프트웨어 개발 단계에서 소요되는 비용을 최소화하려는 연구가 다양하게 진행되고 있다. 프로그램 개발 단계에서는 프로그램을 부분적으로 수정하게 되고 프로그램의 부분 수정인 경우에도 프로그램을 편집하여 다시 실행하는데 많은 시간이 소요된다. 프로그램을 수정할 경우 수정한 부분과 이에 영향받는 부분만을 재실행하기 위한 점진 해석기가 필요하다.

본 논문에서는 프로그램 실행시 동적 의미 분석 방법에 의해 점진 해석기를 구현하고자 한다. IMPLO(IMPerative Language with Object) 언어를 EBNF(Extended Backus Naur Form) 표기법으로 정의하고 동적 의미 구조를 표현하기 위해 작용식(action equation)을 사용하여 IMPLO 언어에 대한 점진 해석기(incremental interpreter)를 설계하여 구현하고자 한다.

Abstract

In order to increase the productivity of software, the research to reduce the total cost in software development environments is working. Considerable time is wasted waiting for a changed program in program development, however small the change, to be edited and compiled and executed. In case of partial change, we need incremental interpreter for reexecuting the changed parts and its affected parts.

In this paper, we implement the incremental interpreter by using analyzing dynamic semantics at execution time. We define a new IMPLO(IMPerative Language with Object) language using EBNF(Extended Backus Naur Form) notation and then, design and implement the incremental interpreter of this language by using action equations to describe the dynamic semantics.

☞ Keyword : Incremental Interpreter, Dynamic Semantics, Action equations, Dependency Chart

1. 서 론

소프트웨어의 생산성을 향상시키기 위해 소프트웨어 개발 단계에서 소요되는 비용을 최소화하려는 연구가 다양하게 진행되고 있다. 프로그램 개발 단계에서는 프로그램을 부분적으로 수정하게 되고 프로그램의 부분 수정인 경우에도 전체 프로그램을 다시 번역하고 실행하여 프로그램을 개발하는 단계에서 많은 시간이 소요된다. 프로그램의 부분 수정인 경우에는 전체 프로그램을 다시 번역하여 재실행하기보다는 수정된 부분과 그 부분에 영향을 받는 부분만을 재실행하는 것이 시간과 공간적인 측면에서 효율적이라 할 수 있다. 프로그램을 수정할 경우, 수정한 부분과 이에 영향받는 부분만을 재실행하기 위해 사용되는 점진 번역 방법은 프로그램 개발 환경에서 실행 효율성 측면에서 고려해 볼 때 매우 중요하다. 이러한 점진 번역 방법을 사용하면 프로그램 전체를 번역하지 않더라도 전체를 번역한 것과 동일한 결과를 얻을 수 있다.

점진 번역기를 구현하기 위해서, 변화된 부분과 그 부분에 영향을 받아 변경되는 부분을 찾아내어 이 부분들을 다시 평가하는 점진 속성 평가 방법[3]을 사용해야 한다. 기존의 점진 속성 평가

* 종신회원 : 협성대학교 경영정보학과 조교수
jlhan@hyupsung.ac.kr(제 1저자)
** 정 회 원 : 남서울대학교 컴퓨터학과 교수
sstar@nsu.ac.kr(공동저자)

방법은 처음 변화된 부분에 영향받는 부분을 찾기 위해 변화를 파급시키는 변화 파급 과정을 거치게 된다. 이 과정은 종속 그래프를 사용하여 아주 복잡하게 수행되며 많은 속성들을 다시 평가해야 하는 문제점이 있다. 즉 종속 그래프는 의미 구조에 직접적인 영향을 주지 않는 속성들의 종속성을 모두 표현하여 이 속성들을 모두 평가하는데 상당한 시간이 소요되며 구조 또한 아주 복잡하다.

본 연구에서는 점진 속성 평가를 위해 종속 차트(dependency chart) 사용 기법[1-3]을 제시한다. 종속 차트는 의미 구조에 직접적으로 영향을 주는 변수의 값을 나타내는 속성을 중심으로 종속성을 표시하여 변화를 추적하는 과정이 효율적으로 수행된다.

속성 문법은 언어의 정적인 의미론을 표현하는 형식적인 표기법으로서 동적인 의미론을 표현하기에는 부적절하다. 동적 의미론을 잘 명세하고 명세된 언어를 구현하기 위해서는 기존의 속성 문법을 확장하고 변형하여 언어 구현에 필요한 동적인 작용들(actions)을 명세해야 할 필요가 있다. 이를 위해 속성 문법을 확장하고 변형하여 정적이고 동적인 의미론을 쉽게 명세할 수 있는 새로운 작용 식(action equation)[1-3]을 제시한다. 제시된 작용식을 기반으로 동적 의미 분석 방법에 의해 점진 번역을 수행하게 되는데 본 연구에서는 번역을 효율적으로 실행되도록 해석 방법을 사용한다. 새로운 IMPLO(IMPerative Language with Object) 언어를 EBNF(Extended Backus Naur Form) 표기법으로 정의하고 제시된 작용식(action equation)의 동적인 의미 구조로부터 IMPLO 언어의 점진 해석기(incremental interpreter)를 구현하고자 한다.

2. 관련 연구

점진 번역기를 구현하는 방법에는 해석 방식을 사용하는 것과 컴파일 방식을 사용하는 방법이 있는데 대부분의 경우 컴파일 기법을 사용해서 점진 번역기를 구현하고 있다.

점진 컴파일러를 구현하는 가장 간단한 방법은 언어에서 컴파일할 수 있는 최소의 단위를 결정하여 소스 코드가 변경되었을 때 그러한 가장 작은 단위를 다시 컴파일 하는 방법이 있다.

컴파일 방법으로 구현된 점진 시스템들 중 ALOE 에디터[11]나 Gandalf 프로젝트[4]에서는 언어의 추상 구문과 관련하여 액션 루틴을 작성하여 점진 컴파일러를 구현하였다. 구문 구동(syntax-directed) 에디터에서 구문 트리를 경유하면서 변화가 생기면 관련된 노드에 대한 액션 루틴을 호출하여 변경된 부분을 다시 평가하게 된다.

점진 컴파일러를 구현하기 위해 속성 문법을 사용했을 경우 속성들간의 종속성(dependency)은 문법에서 자동적으로 유도될 수 있다. 프로그램의 부분 수정인 경우, 속성들간의 종속 정보를 사용해서 주어진 속성 문법에서 최소한으로 필요한 속성이 다시 평가되어 변경된 부분에 영향받는 부분만을 다시 컴파일 하게 된다. 이 방법으로 구현한 대표적 시스템으로 POE 에디터[7]가 있다. POE 시스템[7]은 풀 스크린 언어 기반 에디터(Full-screen Language-based Editor)로 Pascal 언어의 구문적 · 의미적 규칙을 사용하는 환경이다.

통합된 점진 프로그래밍 환경인 Galaxy[9]에서는 변경된 부분에 대한 부트리를 마크하여 마크된 부트리만을 다시 파싱하는 방법을 사용하고 있다. 이 시스템에서는 Galaxy 언어에 대한 점진적 스캐닝과 점진적 파싱을 위해 사용되어질 알고리즘과 방법을 제시하여 시간과 공간적인 면에서 효율적으로 실행되고 있다.

해석 방법을 사용하는 점진 해석기는 프로그램을 수정했을 때 프로그램 전체를 다시 해석하기 않고 변화된 부분과 그 부분에 영향을 받아 변하는 부분만을 해석하여 전체 프로그램을 해석한 것과 같은 결과를 얻어내는 번역 방법을 의미한다. 점진 해석을 수행하기 위해서 속성 문법을 사용하여 속성들간에 종속성을 표현하고 변화가 생

겼을 때 변화된 속성에 종속하는 속성들을 점진 속성 평가 방법을 사용하여 찾아내게 된다. 이 방법으로 구현한 대표적 시스템으로 Cornell Program Synthesizer[13]가 있다.

속성 문법을 사용한 기존의 연구에서는 수정된 부분에서 생긴 변화, 즉 변수 값이 변경될 때 그 변수를 사용한 다른 변수의 값도 바뀌는 영향받는 변수들을 찾아내는 변화 파급(change propagation) 과정을 통해 점진 속성 평가(incremental attribute evaluation)[3]를 수행한다.

본 연구에서는 이러한 복잡한 변화 파급 과정이 필요 없는 종속 차트 기법[1-3]을 사용하고, 소프트웨어를 신속하게 개발할 수 있도록 편리한 사용자 인터페이스를 제공하는 점진 해석기를 구현하고자 한다. 사용자가 쉽게 이해할 수 있는 새로운 IMPLO(IMPerative Language with Object) 언어를 EBNF(Extended Backus-Naur form) 표기법으로 정의하여 이 언어에 대한 점진 해석기를 구축하고자 한다.

3. 점진 해석 수행 방법

점진 해석기를 구현하려면 실제로 언어를 구현하는데 필요한 새로운 작용 식(action equation)[1-3]을 사용하여 동적 의미 구조를 표현하고 동적 의미 분석 방법으로 실행하게 된다. 기존의 연구와 달리 수정된 부분에 대해 변화 파급(change propagation) 과정이 복잡하게 수행되는 점을 개선하기 위해, 속성들간의 종속성을 나타내는 종속 차트(dependency chart)[1-3]를 만들어 수정된 부분과 그 부분에 영향받는 부분을 찾아내어 이 부분들만을 다시 실행하게 된다.

3.1 IMPLO 언어 정의

본 연구에서 번역하여 구현할 IMPLO(IMPerative Language with Object) 언어를 다음과 같이 EBNF (Extended Backus-Naur form) 표기법으로 정의한다.

```
<program>::=<module_list><main_program>
<main_program>::=program <prog_name><decl_
 part><main_body>
<main_body>::=begin <statement_list> end
<module_list>::=<module>{<module>}
<module>::=<sub_program> | <class_module>
<class_module>::=class<class_name>[<deriv
 ed_class>](<formal_list>)<class_body>
<derived_class>::= : parent <class_name>
<class_body>::=begin <class_stmt_list>
               end
<class_stmt_list>::=<class_stmt>{<class_s
 tmt>}
<class_stmt>::=[<modifier>]<statement>
<prog_name>::=<identifier>
<sub_program>::=<sub_heading><sub_body>
<sub_body>::=begin <sub_statement_list>
             end
<sub_heading>::=<sub_keyword><sub_name>
                (<formal_list>)
<sub_keyword>::=procerure | function
<decl_parts>::=<declaration>
                {<declaration>}
<declaration>::=<class_id><identifier>
<class_id>::=<identifier>
<env_id>::=[<env_id_spec>::
          |              <class_id_spec>.]
<identifier>
<env_id_spec>::=<sub_name> | <pro_name>
<class_id_spec>::=<class_name>
<modifier>::=public: | private:
<statement_list>::=<statement>{<statement>}
<statement>::=<in_statement>
            |<out_statement>
            |<ass_statement>
            |<if_statement>
            |<for_statement>
            |<while_statement>
            |<call_statement>
<formal_list>::=<var_list>
<var_list>::=<identifier>{,<identifier>}
<actual_list>::=<actual_parm>
                {,<actual_parm>}
<actual_parm>::=<call_by_reference_symbol>
               <identifier>|<exp_list>
<call_by_reference_symbol>::=&
<sub_name>::=<identifier>
```

```
<sub_statement_list>::=<sub_statement>
                    {<sub_statement>}
<sub_statement>::=<statement>
                  |<return_statement>
<in_statement>::=read <var_list>
<out_statement>::=write <out_list>
<out_list>::=<exp_title_list>
              {,<exp_title_list>}
<ass_statement>::=<env_id> = <exp_list>
<exp_title_list>::=<title_list>
                     |<exp_list>
<title_list>::="<title>"
<title>::=<char>{<char>}
<char>::=<number> | <letter>
          |<special_char>
<exp_list>::=<exp> | <bool>
<call_statement>::=<sub_name>(<actual_list>)
<return_statement>::=return <exp_list>
<if_statement>::=if <condition> then
   <statement_list>[else <statement_list>]
    fi
<for_statement>::=for <identifier> = <exp>
 to <exp>  do <statement_list> od
<while_statement>::=while <condition> do
                      <statement_list> od
<condition>::=<exp><rel_op><exp>
<exp>::=<term>{<weak_op><term>}
<term>::=<element>{<strong_op><element>}
<element>::=(<exp>) | <number> | <env_id>
```

IMPLO 언어는 명령형 언어이면서 객체(object)를 다룰 수 있는 언어이다. 일반적인 명령형 언어의 명령문들인 배정문, if 문, while 문, for 문 및 입출력문을 다루고 있으며, 언어에서 호출할 수 있는 부 프로그램으로는 프로시저(procedure)와 함수(function)가 있다. 부 프로그램에서 매개변수(parameter)를 호출하기 위해 값 호출(call-by-value)과 참조 호출(call-by-reference) 방법이 사용된다. 참조 호출의 경우 호출하는 매개 변수 앞에 '&'를 사용한다.

프로그램에서 주석문을 삽입하여 프로그램의 구문을 설명할 수 있다. 주석문은 '/*' 로 시작하여 '*/' 로 끝난다.

객체(object)를 표현하기 위해 사용자가 정의한 자료형이 클래스이다. 새로운 클래스를 정의할 때는 이 객체를 나타내는데 필요한 자료 부분과 이 객체에 적용할 수 있는 연산 부분을 정의해야 한다. IMPLO 언어에서는 자료를 사용하기 위해 그 자료를 선언할 필요가 없고 배정문으로 바로 사용하면 된다. 주프로그램에서 사용되는 동일한 이름의 자료를 참조하기 위해 '::'를 사용한다. 예를 들면 main::x 는 주프로그램에서 사용되는 변수 x를 의미한다.

IMPLO 언어의 클래스는 공용(public) 파트와 전용(private) 파트로 구성되어진다. 전용 부분은 사용자가 이 자료를 직접 이용할 수 없고 클래스 내에 선언된 멤버 함수를 통해 사용할 수 있다. 공용 파트의 자료를 참조하기 위해서는 다음과 같이 작성한다.

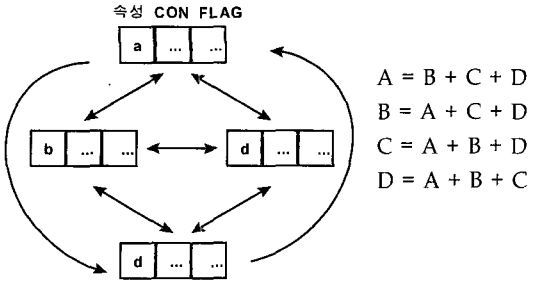
class_name.id

예를 들면 point라는 클래스의 변수 x를 참조하기 위해 point.x라고 표현한다.

3.2 종속 차트

효율적이고 최적의 점진 평가를 수행하는 점진 해석기를 구현하려면 변화된 변수의 값을 속성으로 나타내어 그 변화에 영향받는 속성들을 추적하는 과정이 필요하다. 이러한 속성들의 값을 다시 평가하는 기존의 점진 속성 평가 연구는 종속 그래프(dependency graph)를 사용하여 속성들 간의 종속성을 나타내었고 변화된 속성에 영향받는 속성들을 찾기 위해 복잡한 변화 파급(change propagation) 과정을 거치게 된다. 이런 복잡한 과정을 간단히 하기 위해서 수정된 부분에 대해 변화를 파급시키는 과정이 필요 없는 새로운 점진 평가 방법이 필요하다.

변화 파급 과정이 필요 없는 효율적인 점진 평가 방법을 수행하는 점진 해석기를 구현하려면 프로그램의 명령문들이 변경되었을 때 그 명령문들에 영향받는 부분들을 찾아내는 과정이 필요하



(그림 1) 종속 차트 구조도

고 변수의 값을 나타내는 속성들간의 종속성을 분석하는 방법을 사용하게 된다. 본 연구에서는 새로운 종속 차트(dependency chart)기법[1-3]을 사용하여 수정된 부분에 대해 변화를 파급시키는 과정이 필요 없는 효율적인 점진 해석을 수행하고자 한다.

종속 차트는 종속성을 나타내기 위한 유향(directed) 그래프이다. 각 노드는 속성 이름과 조건 속성을 표시하는 CON 필드와 조건 속성의 형을 나타내는 FLAG 필드의 레코드로 구성되고 아크(arc)는 속성들 간의 종속성(dependency)을 나타낸다.

그림 1의 종속 차트에는 노드의 수가 네 개일 때 발생할 수 있는 모든 가능한 종속성을 아크로 표시하고 있다. 그림의 오른쪽에는 각 속성들의 종속성이 발생할 수 있는 명령문을 예로 들고 있다. 위의 구조도에서는 속성들 간에 발생할 수 있는 모든 종속성을 표시하고 있기 때문에 실제의 프로그램에서는 이 그래프의 아크보다 적은 아크 수의 종속성을 갖게 된다. CON 필드는 조건 속성을 나타내는 것이고 FLAG 필드는 각각의 조건 속성의 형을 구분 짓기 위해 필요한 항목이다.

종속 차트의 경우 속성들 간의 종속성을 나타내는 점은 종속 그래프와 같으나 본 논문에서는 기존의 연구와는 다르게 변수의 값을 나타내는 속성들을 중심으로 속성들 간의 종속성으로 나타내어 값이 변경된 속성의 변화를 파급시키는 과정이 별도로 요구되지 않는다.

3.3 작용 식

어떤 언어에 대한 번역기를 구현하기 위해서는 언어의 동적인 의미 구조를 잘 표현하는 것이 중요하다. 따라서 본 연구에서는 속성 문법을 확장하고 변형하여 정적이고 동적 의미 구조를 표현하는 작용 식(action equation)을 제시함으로써 점진 해석기를 구축하려고 한다.

작용 식에는 Execute equation, Evaluate equation, Eval_rel equation, Eval_par equation, Wait_in equation, 및 Eval_out equation 이 있다.

Execute equation은 IMPLO 언어의 각 명령문을 실행하는 동적 명세를 표현하는 절차적인 식이다. Eval_out equation은 출력문(write 문)에 나오는 변수나 수식의 값을 계산하는 함수적 식이다. Wait_in equation은 키보드로부터 자료가 입력되기를 기다렸다 자료가 입력되면 입력되는 자료 값을 반환하는 함수적 식이다. Evaluate equation은 변수의 값을 나타내는 Value(val) 속성(attribute)을 계산하기 위한 함수적 식으로 계산된 속성 값을 반환하는 식이다. Eval_rel equation은 관계 연산을 평가해서 그 관계식 값을 반환하는 함수적 식이다. Eval_par equation은 괄호를 갖는 수식을 계산해서 그 수식의 값을 반환하는 함수적 식이다.

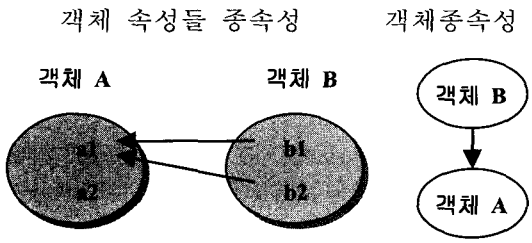
작용 식 중에서 클래스 처리를 위한 Execute equation은 다음과 같다.

```
<class 문의 Execute equation>
·Execute [class <class_name>]→
   class_name.env ← class_name.name
   make_table(class_name.name,
              class_name.env)
   class_name.addr ← current_point
·Execute [class <class_name>
            <derived_class>]→
   class_name.env ← class_name.name
   class_name.penv ← derived_class.name
   make_table(class_name.name,
   class_name.env)
   class_name.addr ← current_point
·Execute [class <class_id><identifier>]→
```

```
identifier.env ← class_id.name
make_table(identifier.name,
           identifier.env)
```

IMPLO 언어에서는 객체들 간에 종속성을 표시하여야 하고 이를 위해 다음과 같은 방법으로 종속성을 나타낸다. A와 B는 객체이고 A의 속성으로 a1, a2가 있고 B의 속성으로 b1, b2가 있고 각 속성들을 다른 객체에서 사용할 수 있는 "public" 특성을 갖는다고 가정할 때 A.a1 = B.b1 + B.b2 명령문에서 객체 A의 속성 a1은 객체 B의 속성 b1과 b2에 종속하게 된다[1]. 즉 객체 B의 속성인 b1이나 b2의 값이 변경될 때 객체 A의 속성 a1의 값도 변하게 된다.

객체들 간의 속성들의 값이 변할 때 속성 평가를 효율적으로 수행하기 위해 객체들간의 종속성을 나타내어야 한다. 객체가 가진 속성들에서 이러한 종속성이 존재할 때 객체들간에 종속성을 그림 2에서처럼 표현한다.



(그림 2) 객체 종속성

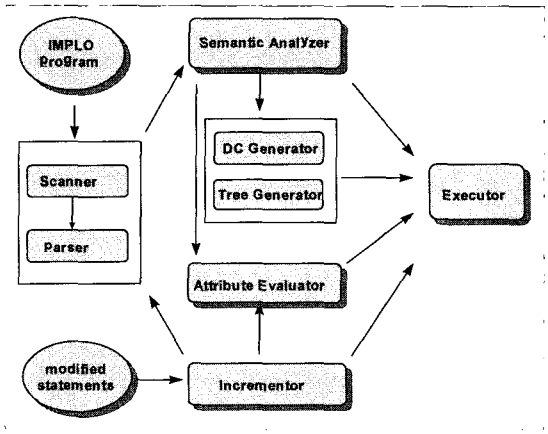
4. 점진 해석기

프로그램의 실행 결과는 각 변수들의 값을 계산하고 저장하여 출력문의 결과를 표시하는 출력 과정으로 나타난다. 폰 노이만 구조에서는 변수의 값을 계산하여 저장하고 그 값을 출력함으로써 정확한 출력 결과를 얻을 수 있다.

점진 실행 시에 정확한 출력 결과를 얻기 위해 의미 구조에 직접적으로 영향을 주는 각 변수의 값을 나타내는 속성(Value attribute) 값을 계산하여 저장해야 한다. 각 변수들은 여러 배정문들을 통해 서로 다른 값이 배정될 수 있다. 라인별로 서로 다른 값을 나타내는 경우를 생각하여 각 변수의 값을 나타내는 속성은 라인별로 값이 변한 기록을 저장한다.

4.1 점진 해석기 구성 요소

점진 해석기는 Scanner, Parser, Semantic analyzer, Attribute evaluator, Executor, Dependency chart generator, Tree generator, Incrementor가 있고, SUN에서 X 윈도우로 사용자 인터페이스를 구현하고 C, Lex, 및 Yacc으로 구현하였다. 본 점진 해석기의 구성 요소는 그림 3과 같다.



(그림 3) 점진 해석의 구성요소

Scanner는 IMPLO 소스 프로그램을 하나의 긴 문자열로 생각하여 차례대로 문자를 검조(scanning) 하여 문법적으로 의미가 있는 일련의 문자인 토큰으로 분할하는 과정을 수행하고 Lex로 구현하였다.

Parser는 스캐너에 의해 생성된 토큰을 받아 IMPLO 언어 문법의 문장이라면 구문 분석 트리를 생성해낸다. IMPLO 문법의 문장이 아니라면 오류 메시지를 주고 오류 교정을 수행한다. 계속해서 다음 문장의 토큰을 입력받아 문법 검사를

수행하는데 파서에 의해 수행되는 구문 분석은 주어진 문자열이 정의된 문법에 의해 생성될 수 있는지 결정하는 과정으로 Yacc로 구현하였다.

Semantic analyzer는 의미 분석을 수행한다. 본 논문에서는 의미 분석을 위해 각 토큰을 별도의 버퍼에 저장하여 프로그램의 첫 번째 토큰부터 마지막 토큰까지 순서대로 의미 분석을 수행하고 그 오류를 검사한다. 별도의 버퍼에 각 토큰을 저장하여 의미적인 구조를 다시 검사하는 이유는 종속 차트(DC)를 효율적으로 생성하고 프로그램의 각 명령문 단위로 트리를 구성하여 변경된 부분에 대한 점진 수행을 효율적으로 실행하기 위해서이다. Semantic analyzer는 각 변수의 Value (val) 속성과 env 속성을 분석하여 형(type)을 검사하고 의미 오류(semantic error) 메시지를 준다. 점진 평가(incremental evaluation)를 수행하기 위해 DC(Dependency Chart) generator를 호출하여 종속 차트(dependency chart)를 생성하고 Tree generator를 호출하여 각 명령문 별로 구문이 저장되는 트리를 생성한다.

DC(Dependency Chart) generator는 종속 차트(dependency chart)를 생성하는 종속 차트 생성기이다. 종속 차트는 종속성을 나타내는 필드와 조건 속성을 나타내는 CON 필드와 필수 속성을 구분 짓기 위한 FLAG 필드가 들어간다. 속성을 빠르게 검색하기 위해 속성들의 테이블인 AT(Attribute Table)가 구성된다[1,2].

Tree generator는 IMPLO 소스 언어를 각 라인 단위로 명령문의 구문이 저장되는 트리를 생성한다. 각 명령문이 소속된 라인을 나타내기 위해 라인 정보가 들어가고 각 라인에 있는 명령문의 구문이 들어가고 다음 명령문을 연결하는 링크가 포함된다.

Attribute evaluator는 의미 분석을 통해 각 변수의 값을 나타내는 Value(val) 속성과 그 외의 여러 속성들을 평가하는 작업을 수행한다.

Executor는 작용 식(action equation)에 의해 명세된 의미 구조에 따라 각 명령문단위로 명령문을 실제로 수행하여 IMPLO 언어의 실행 결과를 화면에 표시한다. 각 명령문 단위로 Execute action에 정의된 동적 의미 구조대로 각 명령문을 수행하게 된다.

Incrementor는 IMPLO 언어의 입력 소스를 수정했을 경우 수정된 명령문만을 어휘 분석하고 구문 분석하여 Tree generator에 의해 생성된 트리의 노드를 대치시키거나 삽입하거나 삭제한다. 수정된 명령문만을 데이터 파일에 저장하여 어휘 분석과 구문 분석을 하게 된다. 프로그램의 각 문장들을 Tree generator를 사용하여 링크트 리스트로 연결하였고 변경된 부분에 대해 트리의 노드를 대치시키거나 삽입하거나 삭제하는 방법을 통해 변경된 부분의 변화를 수정하고 토큰을 저장하는 버퍼의 내용을 역시 수정한다.

점진 해석을 통한 실행 결과를 정확하게 표시하기 위해 변수의 값이 변하게 되는 정보를 라인별로 그 값을 저장하게 된다. 변경된 부분에 대해서만 Executor를 호출함으로써 변경된 병령문을 다시 실행하여 변수 속성 정보 즉 값을 나타내는 속성과 환경을 나타내는 속성을 수정한다. 종속 차트를 통해, 변경된 부분에 대한 실행으로 인해 변화가 발생하는 영향받는 부분을 찾아내어 그 부분을 다시 실행하면 프로그램 전체를 실행하지 않더라도 동일한 결과를 얻을 수 있다. 각 변수는 변수가 속해있는 라인 정보와 속성(val 속성)을 함께 저장하여 그 기록을 남겨 놓아서 동일한 변수에 대해 서로 다른 값을 가질 경우에 각 라인별로 정확한 속성 값을 가질 수 있다.

점진적으로 실행하면서 변수의 속성이 변경될 때 그 변수가 소속된 라인의 변수 속성만을 변경해 주어야 정확한 결과를 얻을 수 있다. 수정되거나 삽입된 명령문에 속한 변수의 속성이 변했을 때 종속 차트를 사용해서 그 속성에 영향받는 속성을 추적하여 다시 그 값을 평가하기 위해 Attribute evaluator를 호출하게 된다. Executor를 호출하여 변경된 변수에 영향받는 속성을 포함한 명령문을 다시 실행한다. 점진 실행 후 프로그램

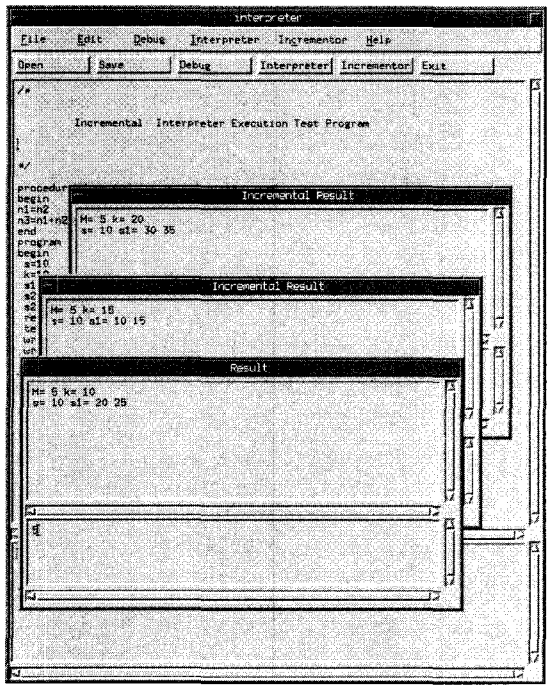
의 결과를 출력하려면 프로그램의 모든 출력문을 수행하여 출력하면 전체를 실행한 것과 동일한 결과를 실행 화면에 표시할 수 있다.

본 연구에서는 점진 해석을 실행하기 위해 편리한 사용자 인터페이스를 구축하였다. 본 사용자 인터페이스는 파일 처리, 에디터, 인터프리터, 점진기(Incrementor)로 구성된다.

파일(File) 메뉴는 파일을 오픈 하거나 저장하는 파일 처리에 관련된 메뉴이고 에디터(Editor)는 스크린 에디터로 일반적인 에디터에서 사용 가능한 Cut, Paste, Copy 등의 서브 메뉴들을 사용할 수 있다. 인터프리터(Interpreter) 메뉴는 전체 프로그램을 실행하기 위해 필요한 메뉴이다. 점진기(Incrementor) 메뉴는 수정된 부분에 대해 점진 실행을 하기 위한 것이다.

점진 해석을 실행하기 위해 먼저 파일을 오픈하여 인터페이스 화면에 그 내용을 올린 다음 Interpreter 메뉴를 선택하여 프로그램 전체를 실행하게 된다. 실행한 후 프로그램의 명령문을 수정했을 경우 그 내용을 다시 저장한 후 Incrementor 메뉴의 부 메뉴 중 Compare 메뉴를 선택하여 수정 전후의 내용을 비교할 수 있다. 비교가 완료되면 Incrementor를 선택하여 수정된 부분에 대해 점진 실행을 수행하게 되고 프로그램을 실행한 결과를 팝업 셀에 출력하게 된다. 프로그램을 수정한 후 사용자는 Interpreter 메뉴와 Incrementor 메뉴 중 하나를 선택하여 프로그램 전체를 실행할 수도 있고 수정된 부분에 대해서만 점진 실행을 선택할 수 있다.

그림 4에는 Interpreter 메뉴로 프로그램 전체를 실행한 팝업 셀과 Incrementor 메뉴를 두 번 사용해서 점진 실행을 수행한 실행 화면을 볼 수 있다. 팝업 셀 중 윗 부분은 실행 결과를 보여주는 윈도우이고 밑부분은 입력된 자료를 보여주는 윈도우이다. 본 점진 해석기의 경우 사용자가 Interpreter 메뉴와 Incrementor 메뉴 중 하나를 선택하여 프로그램을 실행할 수 있으므로 오버헤드가 발생할 수 있는 루프를 수정했을 경우는 Incrementor 메뉴보다는 Interpreter 메뉴를 선별적으로 사용할 수 있다.



(그림 4) 점진 해석기의 비점진과 점진 실행 인터페이스

4.2 실행 효율성 분석

점진 해석기의 실행 효율성을 분석하기 위해 다음 네 유형의 프로그램에 대해 실행 속도를 살펴보았다.

1. 급료를 계산하는 프로그램(P1,P2)
2. 최대 값과 최소 값과 합과 평균을 계산하는 프로그램(M1,M2)
3. 합계를 구하기 위해 프로시저를 호출하는 프로그램(S1,S2)
4. 삼각 함수(sin, cos, tan)를 사용해 각 함수 값을 계산하는 프로그램(I1,I2)

네 프로그램에 대한 점진 해석기의 성능을 표 1에 표시하였다. '전체' 는 수정된 부분만이 아닌

프로그램 전부를 실행한 시간을 의미하고 '점진'은 본 점진 해석기의 Incrementor로 실행한 시간을 의미한다.

(표 1) 네 가지 유형의 프로그램 실행 시간

단위: ms

프로그램 유형	배정문		IF 문		Loop 문		평균		개선 비율
	전체	점진	전체	점진	전체	점진	전체	점진	
P1	6	0.5	7	0.5	8	9	7	3.3	52.9%
P2	12	1	13	1	15	17	13.3	6.3	52.6%
M1	8	0.5	9	0.5	11	12	9.3	4.3	53.8%
M2	13	1	15	1	17	19	15	7	53.3%
S1	11	1	12	1	14	16	12.3	6	51.0%
S2	20	1	22	1	24	27	22	9.7	55.9%
I1	21	1	23	1.5	26	29	23.3	10.5	54.9%
I2	31	2	35	3	39	45	35	16.7	52.3%

표 1을 통해 알 수 있듯이 배정문들이나 IF문을 수정했을 경우, 수정된 부분만을 해석하여 프로그램의 실행 결과를 얻는 시간은 0.5~3ms로 프로그램 전체를 실행한 시간에 비하면 아주 작은 시간이 소요된다. 그러나 루프를 수정했을 경우는 전체 프로그램을 실행한 시간보다 더 많은 시간이 소요된다. 그 이유는 루프의 경우는 점진 평가에 대한 오버헤드까지 포함되기 때문이다. 프로그램을 수정한 후 프로그램 전체를 실행하는 것과 점진 해석을 수행하는 것의 실행 효율성을 비교하여 보면 전체 프로그램을 실행하는 속도를 100%로 생각했을 때 51%~55.9% 속도만큼 빠르게 실행되는 것을 알 수 있다.

5. 결 론

본 연구에서는 언어 기반의 프로그래밍 환경에서 기존의 연구와는 다르게 새로운 종속 차트 기법을 통해 효율적인 의미 분석을 수행하여 프로그램을 변경했을 때 변경된 부분의 변화를 파급시키는 과정이 별도로 요구되지 않았다. 따라서, 본 점진 해석기에서 점진 해석을 수행할 때 효율적인 변화 추적이 가능했다.

프로그램 개발 단계에서 본 해석기에서 생성한 중간 코드 생성 부분을 재 사용할 수 있어 소스 코드가 변경될 때마다 새로운 중간 코드를 생성하는데 소요되는 시간과 공간을 절약할 수 있다. 따라서, 프로그램 개발 단계에서 소스 코드를 중간 코드로 변환하는 과정에서 생성된 중간 코드를 재 사용할 수 있어 소프트웨어 재사용성을 높일 수 있고 소프트웨어 생산성을 향상시킬 수 있다. 또한, 본 해석기를 사용했을 때 IMPLO 언어를 효율적으로 해석할 수 있었다. 특히, IMPLO 언어로 프로그래밍할 수 있는 편리한 사용자 인터페이스를 제공하여 응용 소프트웨어를 작성할 때 유용하게 사용할 수 있다.

네 가지 유형의 프로그램의 실행을 통하여 알 수 있듯이, 배정문이나 IF 문을 수정했을 경우 점진 해석을 수행하는 것이 전체 프로그램을 해석하는 것보다 매우 효율적이었다. 그러나, 루프가 수정된 경우는 수정된 루프 속에 재실행할 필요가 없는 명령문들 수에 따라 실행 효율성이 좌우된다. 평균적으로 점진 해석의 실행 시간을 고려해 볼 때 전체 프로그램을 실행했을 경우보다 약 50% 정도의 속도 개선 효과를 얻을 수 있었다.

참 고 문 헌

[1] 한정란 "작용 식 기반 점진 해석" Ph. D Thesis 이화여대 1999.

[2] 한정란, 이기호 "작용 식 기반 점진 해석기" 정보과학회 논문지 제 26권 8호 pp. 1018~1027 1999.

[3] 이기호, 한정란 "순환 속성 문법의 효율적 점진 평가 기법" 정보과학회 논문지 제 21권 6호 pp. 1116~1126 1994.

[4] A. N. Habermann, "The Gandalf Research project," Computer Science research Review, Carnegie-Mellon University, 1979.

[5] Alfred V.Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1988.

[6] Charles N. F., Greg J. and Jon M., "An Introduction to Editor Allen Poe," Univ. Wisconsin-Madison TR 451, 1981.

[7] Frank G. Pagan, Formal Specification of Programming Language, Prentice-Hall, Inc.

[8] Gail E. Kaiser, "Incremental Dynamic Semantic for Language-Based Programming Environments," ACM Trans. on Prog. Lang. and Sys., Vol.11, No.2, pp.169~193, 1989.

[9] John F. Beetem and Anne F. Beetem, "Incremental Scanning and Parsing With Galaxy," IEEE Transactions on Software Engineering, Vol. 17, No. 7, pp. 641~651, 1991.

[10] Mayer D. Schwartz, Norman M. Delisle and Vimal S. Begwani," Incremental Compilation in Magpie," ACM SIGPLAN Notices pp. 122~131, 1984.

[11] Raul Medina Rora and David S. Notkim, "ALOE users' and implementers' guide, " Carnegie-mellon Computer Science Department Research Report CS-81-145, 1981.

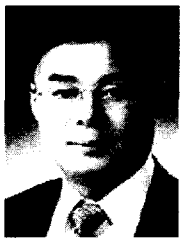
[12] Steven P. Reiss, "An approach to Incremental Compilation," ACM SIGPLAN Notices pp. 144~151, 1984.

[13] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizes: A Syntax-directed Environment," Communication ACM, Vol 24(9), pp. 563~573, 1981.

◐ 저 자 소 개 ◑

한 정 란

1985년 이화대학교 전자계산학과 졸업(학사)
1987년 이화대학교 대학원 전자계산학과 졸업(석사)
1999년 이화대학교 대학원 컴퓨터공학과 졸업(박사)
1999~현재 협성대학교 경영정보학과 교수
관심분야 : e-Commerce, XML, e-Business, e-CRM, 점진 번역기 등
E-mail : jlhan@hyupsung.ac.kr



최 성

강원대학교 대학원 컴퓨터과학과 졸업(박사)
연세대학교 산업대학원 전자계산학과 졸업(석사)
동국대학교 산업시스템학과 졸업(학사)
전 한국생산성본부 국장, 제주은행전산실장
현재 남서울대학교 컴퓨터학과 교수
관심분야 : EC/ERP, VR영상게임, 소프트웨어공학
저서 : 비즈니스리엔지니어링, ERP시스템기초 등 33권
E-mail : sstar@nsu.ac.kr