DOI: 10.3745/KIPSTA.2010.17A.4.173

예외 처리를 위한 형식 의미론

한 정 란†

요 약

프로그래밍언어에 대한 형식 의미를 적절하게 표현하면 설계와 표준화, 최적화 및 번역 과정에 중요한 역할을 수행한다. 기존의 자바에 대한 형식 의미 연구는 번역을 위해 정확하고 실제적인 의미 구조를 표현하는데 미흡한 점이 있다. 자바 번역 과정의 효율성을 향상시키기 위해 정적이고 동적인 의미 구조를 정확하게 표현하는 의미 표현법에 관한 연구가 필요하다.

본 논문에서는 기존의 작용식(Action Equations)을 사용해 명세한 연구를 확장하여 자바 언어에 대한 형식 의미를 명세하는 개선된 작용식(Action Equations)을 제시한다. 객체를 다루는 기능은 물론 예외를 처리하는 형식 의미론을 보다 실제적이고 정확하게 명세하게 된다. 판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility)의 네 영역에서 명세된 작용식을 기존의 의미 표현법과 비교하여 본 작용식의 우수성을 확인하고자 한다.

키워드 : 형식 의미론, 작용식, 의미구조 명세, 예외처리 명세

Formal Semantics for Processing Exceptions

Han, junglan†

ABSTRACT

To specify a formal semantics is to do a significant part for design, standardization and translation of programming languages. The existing studies of a formal semantics for Java have a weak point to describe a clear and practical semantics for an efficient translation. It is necessary to do research for a formal semantics to specify a static and dynamic semantics clearly in order to do an efficient translation.

This paper presents the improved Action Equation that specifies a formal semantics for Java to extend the research using Action Equation. The Action Equation is a practical and accurate specification that describes object-oriented programming features and handles exceptions. The specified Action Equation is compared to other descriptions, in terms of readability, modularity, extensibility, and flexibility and then we verified that Action Equation is superior to other formal semantics.

Keywords : Formal Semantics, Action Equation, Specification of Semantics, Specification of Exception Handling

1. 서 론

소프트웨어의 대형화 추세에 따라 객체 지향 프로그래밍이 중요한 패러다임으로 인식되고 있다. 통합개발도구(IDE)나 라이브러리, 소프트웨어 개발 방법론 등의 일련의 컴퓨터 소프트웨어 개발 도구들이 객체 지향적 특징을 지원하는 방향으로 발전하고 있어 객체 지향 프로그래밍이 각광받고 있다. 객체 지향 프로그래밍은 유연하고 변경이 용이한 프로그램을 작성할 수 있어 대규모 소프트웨어 개발에 많이 사용된다. 소프트웨어를 개발하고 유지하고 보수하는 측면에서 객체 지향 프로그래밍의 장점이 부각되어 객체 지향 언어가 많이 사용되고 있고 자바는 널리 사용되는 객체 지향 언어 중의 하나이다.

어떤 언어에 대한 형식 의미를 표현하는 것은 그 언어의 설계와 표준화, 최적화 및 번역 과정에서 중요한 역할을 수행하므로 자바에 대한 형식 의미 연구는 표준화나 번역을 위해 의미 있는 연구 분야라 할 수 있다. 기존에 연구된 여러 형식 의미론들은 번역 과정에 필요한 동적 의미구조를 구체적이고 정확하게 기술하고 있지 않아 최적화나 번역 과정에 실제적인 도움을 주는 데 미흡한 점이 있다. 효율적인 번역과정을 수행하기 위해 보다 구체적이고 실제적인 의미 구조를 표현하는 형식 의미 명세에 관한 연구가 필요하다. 객체지향 언어로 각광받고 있는 자바의 정적 의미 구조와 동적인 의미 구조를 실제적이고 정확하게 표현하는 형식 의

† 종신회원 : 협성대학교 경영정보학과 부교수
논문접수 : 2010년 3월 4일
수 정 일 : 1차 2010년 4월 26일, 2차 2010년 5월 19일
심사완료 : 2010년 5월 19일

미 표현법에 대한 연구는 중요한 분야라 할 수 있다.

본 논문에서는 기존에 간단한 객체지향언어인 OBLA(Object Language) 언어에 대한 의미 구조를 표현한 작용식(Action Equations)[1]을 확장하여 자바 언어에 대한 형식 의미를 명세하는 작용식(Action Equations)을 제시한다. 작용식을 확장하고 추가하여 자바언어에서 객체를 다루는 기능이나 예외를 처리하는 기능을 명세하여 보다 실제적이고 정확한 형식 의미 구조를 표현하게 된다.

판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility)의 네 영역에서 명세된 작용식을 기존의 의미 표현법과 비교하여 본 작용식의 우월성을 확인하고자 한다.

본 논문의 구성을 살펴보면 2장에서는 형식 의미론과 관련된 기존 연구들에 대해 소개하고 있다. 3장에서는 기존의 연구에서 확장하여 객체를 처리하는 구문에 대해 작용식을 새롭게 제시하고 있고 4장에서는 예외를 처리하기 위한 자바 구문에 대해 작용식을 사용하여 형식 의미를 명세하고 있다. 5장에서는 기존의 자바에 대해 명세한 의미 표현법과 제시한 작용식을 비교하여 평가하고 있고 6장에서 제시한 작용식의 우수성에 대해 결론짓고 있다.

2. 관련 연구

객체 지향 언어와 연관된 여러 형식 의미론에 관한 연구가 진행되고 있는데 구현과 관련된 언어 자체에 대한 의미론을 명세하는 연구가 있고 객체 지향 모델과 관련된 의미론을 명세하는 연구들이 있고 그래픽 기반의 명세 연구들이 있다.

도메인-특정 의미론(Domain-specific Semantics) 연구[5]에서는 객체 모델에 관한 형식 의미론을 제시한 것으로 클래스 invariants 와 연산 명세에 포함된 전역 제한(global constraints)과 결합(association) 제한을 다루고 있다. 특별한 구현방법을 반영하는 변환을 개발하는데 객체 모델을 위한 도메인 한정 의미론이 어떻게 사용될 수 있는 지 보여주고 있다[5].

그래픽 기반 의미론(Graph-based Semantics) 연구[6]에서는 객체 지향 클래스 구조 명세를 위한 그래프 기반 형식론을 제시하는데 클래스 모델 그래프(class-model graph)라 불리는 방법론에서 객체들 간에 단일 상속 관계와 확장 클래스 안에 있는 메서드들 간에 재정의(overriding) 관계를 라벨화된 그래프로 결합하고 있다. 노드들 위의 관계는 객체 간에 상속 관계를 만드는데 사용하고 간선(edges)위의 관계는 확장된 클래스들안에 메서드를 재정의하는데 사용하고 있다[6].

객체 지향 언어를 위한 의미 명세 연구[1]에서는 객체 지향 언어인 OBLA(Object Language) 언어를 EBNF(Extended Backus Naur Form) 표기법으로 새롭게 정의하고 이 언어의 객체 지향 특성을 처리하는 작용식(action equation)을 구체적이고 명확하게 명세하고 있다[1].

객체 지향 언어의 의미를 정의한 논문들 중에서 자바에 대한 의미 표현을 정의한 연구들[7, 8]을 살펴보면 크게 세 가지 표현법으로 분류할 수 있다: Alves-Foss와 Lam의 Denotational Semantics(DS) 표현, Borger and Schulte의 Abstract State Machine(ASM) 표현, Watt 와 Brown의 Action Semantics(AS) 표현.

DS(Denotational Semantics) 표현은 통상적인 λ-표기법을 사용하고 연속 전달(continuation passing) 방식으로 작성되어 각 구문 구조의 의미가 고급 명령(Higher-order) 함수에 의해 표현되어진다. 자바에 대한 DS 표현은 오류가 많고 잘 정의되어있지 않아 자바를 완전하게 표현하기 힘들다[7].

ASM(Abstract State Machine) 표현은 부 표현(sub description)의 시리즈로 구성되고 고도의 단순화된 추상 구문으로 상태들 사이에 전이를 통해 상태 공간을 정의하여 작동하고 있고 이 상태는 복잡한 구조를 갖는다. ASM 표현은 자바의 전체 구문으로 확장 가능하지만 많은 구조를 처리하기에는 불완전한 측면이 있다[7]. ASM은 자바의 실제 구문과는 다른 추상구문으로 매우 명확하지만 저급 수준이고, 제한된 의미에서만 모듈방식으로 사용된다[7].

AS(Action Semantics) 표현은 스레드와 메소드 중복(overloading)을 제외한 모든 자바 언어를 표현할 수 있고 모듈의 집합으로 구조화되어 있다. AS 표현은 DS의 모듈화를 개선한 것으로 동작(action)이라 불리는 표시(denotations) 형식을 취하고 있고 프로그래밍 언어를 구현할 수 있도록 구체적으로 표현되어 있지 않아 실제로 번역기를 만들 때 어려움이 있다[1].

본 연구에서는 기존에 OBLA(Object Language) 언어에 대한 의미론을 작용식(Action Equations)[1]을 사용해 명세한 연구를 확장하고 추가하여 객체 지향 언어로 널리 보급되어 있는 자바 언어에 대한 의미구조를 향상된 작용식(Action Equations)을 통해 표현하고, 다른 기존의 연구들보다 구체적이고 확장 가능하고 융통성을 높여 번역기를 쉽게 구현할 수 있는 정적·동적 의미구조를 표현하는 것을 목표로 한다. 특히, 클래스를 처리하는 전반적인 객체 지향 특징은 물론 프로그램의 신뢰성을 증진하는 예외 처리에 대한 형식 의미를 중점적으로 명세하고자 한다.

3. 작용식

언어에 대한 형식 의미를 표현하는 것은 그 언어의 설계와 표준화, 최적화 및 번역 과정에서 중요한 역할을 수행한다. 어떤 언어를 구현하는 과정을 효율적으로 수행하려면 그 언어의 의미구조를 정확하고 구체적으로 명세하여야 한다. 객체 지향언어로 널리 사용되는 자바 언어에 대한 동적 의미구조를 정확하게 표현함으로써 자바 언어를 구현하는 과정에 사용될 형식 의미들을 명확하게 정의하여 번역기를 손쉽게 구현할 수 있다. 따라서 본 논문에서는 기존에 연구된 작용식[1]을 확장하고 기능을 추가하여 자바에 대한 형식 의미를 새롭게 명세하고자 한다.

작용식 중에서 각 명령문을 실행하는 동적 명세를 표현하

고 순차적으로 실행되는 절차적인 작용(action)인 Execute equation을 사용하여 자바언어를 다루기 위한 동적 의미구조를 표현한다. 기본적인 명령문들은 문법에 차이가 있을 뿐 형식 의미상에 차이는 없고 기존 연구와 유사하게 정의할 수 있어 자바에서 추가된 객체 처리와 관련된 명령문들을 중점적으로 다루고자 한다.

작용식에서 사용하는 속성들에는 **out, val, env, pri, name, addr, penv** 등이 있다[1]. **out** 속성은 각 작용식을 실행한 후에 발생하는 결과를 나타내는 합성(synthesized) 속성이고 **val** 속성은 각 비단말(nonterminal)의 값을 나타내는 합성 속성이다. **env** 속성은 비단말의 환경을 나타내는 것으로 메소드를 호출하는 환경과 호출되는 환경을 구분하기 위한 전이(inherited) 속성이고 **pri** 속성은 수식에서 연산자의 우선 순위를 지정하기 위한 속성이다. **name** 속성은 식별자의 이름을 나타내는 속성이고 **addr** 속성은 메소드를 호출하기 위해 주소를 저장하는 속성이고 **penv** 속성은 하위 클래스에서 상위 클래스의 환경을 물려받기 위해 상위 클래스의 환경을 저장하는 속성이다. 자바언어에서는 객체를 처리하는 선언문이나 자료를 선언할 때 해당 자료형을 표시하고 있어 자료형을 저장하는 부분으로 **type** 속성을 추가하여 사용한다.

각 작용식(action equation)에서 필요한 함수나 절차적인 수행을 필요로 하는 모듈에는 lookup, print_out, repeat, return_save, control_transfer 모듈 등이 있다[1]. **lookup** 모듈은 변수 이름을 받아서 그 변수에 저장된 값을 반환하는 함수이고 **print_out** 모듈은 변수나 수식의 값을 화면에 출력하는 함수이다. **repeat** 모듈은 작용식에서 반복적으로 수행되는 작용을 표현하는 모듈로 repeat_start가 나오면 처음부터 다시 수행하고 repeat_end가 나오면 반복 수행을 끝낸다. **return_save** 모듈은 메소드를 호출할 때 실행하는 모듈로 호출되는 메소드의 반환 주소를 저장한다. **control_transfer** 모듈은 호출하거나 반환될 때 프로그램의 제어를 이동시키는 모듈이다. **determine_environment** 모듈은 식별자 이름이 주어졌을 때 그 식별자의 환경을 반환하는 모듈로 식별자가 클래스에 속한 경우 클래스 이름을 반환하고 메소드 안에 선언된 경우 메소드 이름을 반환한다. **search_member** 모듈은 클래스 이름이 주어졌을 때 그 클래스안에 선언된 멤버 변수들을 반환하는 모듈이다.

자바의 의미구조를 명세하기 위해 세가지 모듈인 **make_table, make_table2, lookup2** 모듈을 추가로 정의하여 객체처리와 관련된 작용식을 표현한다. 심벌테이블을 만들 때, 자료형을 저장하는 부분으로 메서드에서 사용된 형식 매개변수를 심벌테이블에 저장하기 위해 **make_table** 모듈을 사용한다. **make_table2** 모듈은 객체와 관련된 정보를 저장하기 위한 것으로 새로 정의한 모듈이다. 형식 매개변수를 정의할 때 선언된 자료형을 찾기 위해 **lookup2** 모듈을 사용하여, 이미 선언되어 저장된 자료형을 심벌테이블에서 찾을 수 있다.

lookup2 모듈은 변수 이름을 받아서 그 변수의 자료 형을 반환하는 함수이다. **make_table** 모듈은 메소드에서 선언된 형식 매개변수에 관한 정보를 심벌테이블에 만드는 모듈로 각 매개변수의 이름과 메소드의 이름과 자료형과 환경을 사용해서 매개변수에 대한 정보를 구성한다. **make_table2** 모듈은 클래스와 관련된 정보를 심벌테이블에 만드는 모듈로 각 클래스 이름과 환경을 사용해서 정보를 구성한다. 각 클래스에 대해 한번만 실행되는 모듈이다.

자바언어의 객체 처리와 관련된 작용식을 명세하면 (그림 1)과 같다.

```
◀Execute [stmts] → event [ complete | diverge ]
Execute [<s1>,<s2>,... <sn> where n ≥1] →
  s1.out ← Execute [<s1>]
  s2.out ← Execute [<s2>]
  ...
  sn.out ← Execute [<s2>]
Execute[<method_name>(<var_type1><param1>,<var_type2><param2>,..., <var_typen><paramn>); where n ≥1] →
    for i=1 to n do
        parami.val ← lookup(method_name.env, method_name.name)
        parami.type ← var_typei.name
        make_table(parami.name, parami.type, method_name.env)
    od
    method_name.addr  ← current_point
Execute [<method_name>(<param1>, <param2>, ..., <paramn>); where n ≥1] →
    for i=1 to n do
        parami.env ← method_name.env
        parami.val ← Eval_out[<parami>]
    od
    return_save(method_name.addr)
    control_transfer(method_name.addr)
Execute [class <class_id>]→
    class_id.env ← class_id.name
    make_table2(class_id.name, class_id.env)
    class_id.addr  ← current_point
```

(그림 1) 객체 처리 작용식

```
·Execute [public class <class_id>]→
      class_id.env ← public
      make_table2(class_id.name, class_id.env)
      class_id.addr  ← current_point
·Execute [class <class_id> extends <derived_class>]→
      class_id.env ← class_id.name
      class_id.penv ← derived_id.name
      make_table2(class_id.name, class_id.env)
      class_id.addr  ← current_point
·Execute [public class <class_id> extends <derived_class>]→
      class_id.env ← public
      class_id.penv ← derived_class.name
      make_table2(class_id.name, class_id.env)
      class_id.addr  ← current_point
·Execute [<class_id><object_id1>,<object_id2>,...<object_idn>; where n ≥1]→
      for i=1 to n do
      object_idi.env ← class_id.name
      make_table2(object_idi, object.env)
      od
·Execute [<modifier><var_type><identifier1>,<identifier2>,<identifiern>; where n≥1 ]→
      for i=1 to n do
        identifieri.name ← identifieri
        environment.name ← determine_environment(identifieri.name)
        identifieri.env ← environment.name
        identifieri.type ← var_type.name
        make_table(identifieri.name, identifieri.type, identifieri.env)
      od
·Execute[<object_id> = new<class_id>(<param1>,<param2>,...,<paramn>) where n≥)]→
      object_id.env ← class_id.name
      make_table2(object_id.name, object_id.env)
      member_identifier  ← search_member(class_id.name)
      member_identifier.type ← lookup2(method_name.env, method_name.name)
      for each member_identifier do
         make_table(member_identifier.name, member_identifier.type, object_id.name)
      do
      object_id.addr  ← current_point
      for i=1 to n do
        parami.env ← class_id.env
        parami.val ← Eval_out[<parami>]
      od
      return_save(class_id.addr)
      control_transfer(class_id.addr)
·Execute [<class_id><object_id> = new <class_id>(<param1>, <param2>, ..., <paramn>)
where n ≥1)]→
      object_id.env ← class_id.name
      make_table2(object_id.name, object_id.env)
      member_identifier  ← search_member(class_id.name)
      member_identifier.type ← lookup2(method_name.env, method_name.name)
      for each member_identifier do
         make_table(member_identifier.name, member_identifier.type, object_id.name)
      do
      object_id.addr  ← current_point
      for i=1 to n do
        parami.env ← class_id.env
        parami.val ← Eval_out[<parami>]
      od
      return_save(class_id.addr)
      control_transfer(class_id.addr)
```

(그림 1) 의 계속

4. 예외 처리

자바에서 예외를 정의하기 위해 Throwable 클래스나 그 확장클래스중의 하나로부터 확장된 예외 객체를 생성하고 보통의 경우 Throwable 클래스의 확장클래스인 Exception을 확장하여 새로운 예외를 처리하는 클래스를 만든다. Throwable 클래스에는 예외가 일어난 상황을 설명하는 여러 가지 출력메시지들을 포함하고 있다[4].

Object 클래스의 확장 클래스로서 Throwable 클래스가 존재하고 그 확장클래스로 Error와 Exception 클래스가 존재한다. Error 클래스는 정상적인 프로그램에서 감당할 수 없는 심각한 오류를 나타내고 이와 같은 오류는 프로그래머

가 처리하지 않더라도 자바시스템에서 자동으로 처리한다. Exception 클래스는 정상적인 프로그램 실행 과정에서 발생할 수 있는 예외를 의미하고 필요에 따라 프로그래머가 예외처리기를 작성하여 처리할 수 있다[4]. 자바에서 throw 구문을 사용해서 예외를 발생시키고 그에 해당하는 적절한 예외처리기를 실행할 수 있다.

예외를 처리하는 작용식을 정확하게 명세하기 위해 세 가지 모듈 **make_exception_table, lookup_excep, lookup_addr**을 새롭게 정의하여 사용하고 예외를 처리하기 위한 작용식을 보다 구체적이고 실제적으로 명세한다. **make_exception_table** 모듈은 **throws** 구절을 통해 발생 가능한 예외를 명시했을 때 그 예외들의 목록인 **exception_table**을 작성하는 모듈로 발생 가능한 예외가 처리될 주소를 **catch** 구문에서 찾아 기록하는 모듈이다. **exception_table**은 **예외이름**, **라인정보**, **예외발생** 및 **예외처리**의 네 필드로 구성된다. **예외이름**은 발생 가능한 예외들을 기술하는 필드이고, **라인정보**는 예외를 처리할 catch 구문의 라인 번호를 나타내고, **예외발생**은 예외 발생 여부를 나타내는 필드이고, **예외처리**는 발생한 예외의 처리여부를 나타내는 필드이다. **lookup_excep** 모듈은 시스템에서 내장된 예외와 **exception_table** 목록에서 예외를 찾아 그 예외가 발생했는지를 반환하는 모듈이다. **lookup_addr** 모듈은 **exception_table**에서 발생한 예외를 찾아 예외가 처리될 프로그램의 라인 번호를 가져와서 예외를 처리하기 위해 필요한 위치정보(주소)를 반환하는 모듈이다. 프로그램의 라인정보를 통해 그 라인의 상대주소를 계산할 수 있고 프로그램의 시작 주소를 알면 실행시 절대주소를 계산할 수 있다.

(그림 2)의 자바 구문에 대해 **exception_table** 목록을 작성하는데, 편의상 각 명령문에 라인번호를 나타내고 있고 첫 명령문인 **try** 구문이 10라인에서 시작하는 것으로 가정한다.

<표 1>에서 볼 수 있듯이 **예외이름** 필드를 통해 발생 가능한 예외이름을 알 수 있고 **라인정보** 필드로 그 예외를

```
10:  try {
11:    //...
12:   }catch(ExceptionType1 identifier) {
13:    //...
14:   }catch(ExceptionType2 identifier) {
15:    //...
16:    :
17:   }catch(ExceptionTypen identifier) {
18:    //...
19:   }finally{
20:    //...
21:   }
```

(그림 2) 예외 처리 자바 프로그램

〈표 1〉 exception__table 목록

예외 이름	라인 번호	예외발생	예외 처리
$ExceptionType_1$	12	false	false
$ExceptionType_2$	14	false	false
…	…	false	false
$ExceptionType_n$	17	false	false

처리하는 **catch** 구문의 라인번호를 찾을 수 있다. (그림 3)의 **throw new** 작용식 안의 **excep_class_id.addr ← lookup_addr(excep_class_id.name)** 에 의해 **exception_table**의 라인정보로 실제 절대주소 값을 계산하여 excep_class_id.addr 에 예외를 처리할 주소 값이 들어간다. **예외발생** 필드는 throw new 구문으로 예외가 발생하면 true값을 갖고 초기값은 false이다. (그림 3)의 **throw new** 작용식 안의 **excep_class_id.val ← true** 로 excep_class_id.val은 true가 되어 예외가 발생한 것을 나타낸다. (그림 3)의 **throw new** 작용식 안의 **if (excep_class_id.val) then control_transfer (excep_class_id.addr)**에 의해 예외를 처

```
·Execute[<method_name>(<var_type1><param1>,<var_type2><param2>,..., <var_typen><paramn>) where n ≥1
throws <excep_id1>,<excep_id2>, ... <excep_idm> where m ≥1] →
    for i=1 to n do
        parami.val ← lookup(method_name.env, method_name.name)
        parami.type ← var_typei.name
        make_table(parami.name, parami.type, method_name.env)
    od
    method_name.addr  ← current_point
    for i=1 to m do
      make_exception_table(excep_idi.name)
    od
·Execute[throw new <excep_class_id>(<param1>,<param2>,...,<paramn>) where n≥1]→
    anonym_id.env ← excep_class_id.name
    anonym_id.type ← lookup2(anonym_id.env, anonym_id.name)
    make_table(anonym_id.name, anonym_id.type, anonym_id.env)
    member_identifier.name ← search_member(excep_class_id.name)
    for each member_identifier do
       make_table(member_identifier.name, member_identifier.type anonym_id.name)
```

(그림 3) 예외 처리 작용식

```
      do
      anonym_id.addr  ← current_point
      for i=1 to n do
        param_i.env ← excep_class_id.env
        param_i.val ← Eval_out[<param_i>]
      od
      return_save(excep_class_id.addr)
      if (lookup_excep(excep_class_id.name)) then
         excep_class_id.val ← true
         excep_class_id.addr ← lookup_addr(excep_class_id.name)
      endif
      if (excep_class_id.val) then
         control_transfer(excep_class_id.addr)
      else control_transfer(NullPointerException.addr)
      endif
·Execute[try { <s1>,<s2>,... <sn> where n ≥1 }<catch_stmts>]→
   Execute[<s1>,<s2>,... <sn> where n ≥1]
   if (lookup_excep(excep_class_id.name)) then
      Execute[<catch_stmts>]
·Execute[try { <s1>,<s2>,... <sn> where n ≥1 }<catch_stmts><finally_stmt>]→
  Execute[<s1>,<s2>,... <sn> where n ≥1]
   if (lookup_excep(excep_class_id.name)) then
      Execute[<catch_stmts>]
   else Execute[<finally_stmt>]
·Execute[<finally_stmt>] →
  Execute[finally {<s1>,<s2>,... <sn> where n ≥1 }]
·Execute[finally {<s1>,<s2>,... <sn> where n ≥1 }]→
  Execute[<s1>,<s2>,... <sn> where n ≥1]
·Execute[<catch_stmts>] →
  Execute[catch_i (<excep_class_id_i><excep_id_i>){<s1>,<s2>,... <snm>where nm≥1} where i≥1]
·Execute[catch_i (<excep_class_id_i><excep_id_i>){<s1>,<s2>,... <snm>where nm≥1} where i ≥1]
→ case of excep_class_id.name
    'excep_class_id1.name':
      Execute[catch (<excep_class_id1><excep_id1>){<s1>,<s2>,... <sn1>where n1≥1}]
    'excep_class_id2.name':
      Execute[catch (<excep_class_id2><excep_id2>){<s1>,<s2>,... <sn2>where n2≥1}]
      ....
    'excep_class_idn.name':
      Execute[catch (<excep_class_idn><excep_idn>){<s1>,<s2>,... <snm>where nm≥1}]
   endcase
·Execute[catch (<excep_class_id><excep_id>{<s1>,<s2>,... <sn> where n ≥1}] →
  Execute[<s1>,<s2>,... <sn> where n ≥1]
```

(그림 3)의 계속

리하는 주소로 제어가 이동된다. (그림 3)의 **try** 작용식에서 **Execute[<s_1>, <s_2>,... <s_n> where n ≥1]** 에 의해 **try** 구문의 작용식을 수행하고, **if (lookup_excep(excep_class_id.name)) then Execute[<catch_stmts>]** 에서 예외가 발생하면 해당 **<catch _stmts>**의 작용식을 수행하여 예외를 처리하고 발생한 예외가 처리되면 **예외처리** 필드에는 true 값을 갖게 된다.

5. 형식 의미 표현법의 비교

기존의 형식 의미 표현법과 작용 식을 비교하기 위해 예외 처리를 위한 구문 중 “throw” 문에 대한 형식 의미를 정의한 **DS(Denotational Semantics), ASM(Abstract State Machine), AS(Action Semantics)** 세 가지 표현법을 기존 연구 논문[9]에서 발췌하여 나타내면 다음과 같다.

- **DS(Denotational Semantics) 표현법**

exec ⟦ **throw** Expr ; ⟧ env scont sto =
eval ⟦ Expr ⟧ env econt sto **where**
econt = λ(val, typ, sto_1).$scont_1$ (env_2, sto_1) **where**
env_2 = env_1[&thrown ← (val, typ)] and
$scont_1$ = env[&thrown]

- **ASM(Abstract State Machine) 표현법**

if task is (throw exp ;) **then**
if val(exp) ≠null **then**
mode := Throw(val(exp))
task := up(task)
else
fail(NullPointerException)

• AS(Action Semantics) 표현법

```
Execute [ "throw" E:Expression ]=
   evaluate E then
      check ( the given reference is not null) and then
      escape with the throw of the given value
   or
      check ( the given reference is not null) and then
      escape with the throw of the null-pointer-exception
```

형식 의미를 비교하기 위해 기존 연구[7-9]들을 중심으로 <표 2>에 판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility) 영역에서 세 표기법인 DS, ASM, AS와 제시된 작용식(AE)을 비교하였다. 기존의 세 가지 표기법에 대한 판정은 형식 의미에 대해 비교하고 연구한 기존 논문[7, 9]을 근거로 표시하였다.

판독성, 모듈성, 확장성, 융통성의 네 영역에서 작용식을 비교하면 먼저, 판독성의 경우, **ASM**이나 **AS**와는 달리 동적 의미구조를 표현할 때 **if**나 **for**같은 일반적으로 널리 알려진 제어 구문을 사용하고, 변수 값이나 자료형, 환경 등의 8가지 속성을 사용하여 동적 의미구조를 정확하고 자세하게 표현하여 구문이 뜻하는 동적 의미구조를 쉽게 파악할 수 있어 판독성이 뛰어나다. (그림 4)는 **throw new** 작용식에서 예외처리 발생 객체의 매개변수에 대해 처리하는 의미구조로, 다른 세 표현법과는 달리 **env** 나 **val** 속성을 사용하여 매개변수의 환경과 매개변수 값을 정확하고 구체적으로 표현하고 구현에 필요한 모든 의미구조에 대한 값을 명백하게 정의하고 있어 매개변수의 자료형과 관련된 의미 구조를 쉽게 이해할 수 있어 판독성을 높이고 있다.

모듈성을 고려해보면 모듈성이 높은 **AS**와 비교할 때, **AE**의 경우 자바에 대한 형식 의미를 보다 구체적이고 정확하게 명세하기 위해 lookup, print_out, repeat, return_save, control_transfer, determine_environment, search_member, lookup2, make_table, make_table2, make_exception_table, lookup_excep, lookup_addr 의 13가지 모듈을 사용하고 있고 번역 과정을 수행할 때 각 기능을 수행하는 모듈을 메소드로 작성하여 구현함으로써 모듈성을 향상시킬 수 있다.

〈표 2〉 형식 의미 비교

표현법	판독성	모듈성	확장성	융통성
DS	낮음	낮음	낮음	보통
ASM	보통	보통	높음	높음
AS	보통	높음	높음	낮음
AE	높음	높음	높음	높음

```
for i=1 to n do
    parami.env ← excep_class_id.env
    parami.val ← Eval_out[<parami>]
od
```

(그림 4) 매개변수 처리 의미구조

확장성이 높은 **ASM**이나 **AS**와 비교할 때, 본 논문에서 볼 수 있듯이 **AE**의 경우 OBLA 언어 같은 간단한 객체 처리를 위한 작용식[1]뿐만 아니라 기능이 복잡한 자바의 객체 처리나 예외 처리 등을 포함하여 언어의 기능을 확장했을 때, 다른 두 표현법과는 달리 본 논문에서 명세한 것처럼 make_table, make_table2, lookup2, make_exception_table, lookup_excep, lookup_addr 모듈 같은 추가의 모듈을 정의하여 확장된 기능에 대한 동적 의미구조를 정확하게 명세할 수 있다. 작용식은 간단한 객체 지향언어[1]에 대해 형식 의미를 명세할 수 있고 본 논문에서 명세했듯이 자바와 같은 복잡한 객체 지향 언어의 의미를 표현할 수 있어 확장가능성이 높은 형식 명세법이다.

융통성이 높은 **ASM**과 비교할 때, 간단한 객체 지향언어인 OBLA 언어[1]뿐만 아니라 다른 객체 지향 언어 구문을 명세할 경우 작용식의 문법 구문만 변경하여 그 언어의 동적 의미구조를 쉽고 적절하게 명세할 수 있어 융통성 있는 형식 의미 명세법이라는 사실을 알 수 있다. 또한 프로그램간에 의미 차이가 있을 경우 **ASM**과는 달리 8가지 속성을 사용하여 의미를 보다 명확하게 전달할 수 있으므로 정의된 의미 구조를 구현하는 과정이 더 효율적으로 진행될 수 있다.

제시된 작용 식은 연산 의미론(operational semantics)을 근거로 변수 값이나 자료형, 환경 등의 8가지 속성을 사용하여 간단하고 쉽게 정적의미뿐만 아니라 동적 의미구조도 표시하고 있다. 형식 의미를 정확하게 정의하는 목적이 최적화나 번역기를 구현하는 과정에 사용하는 것인데 본 작용식을 사용하여 구현된 번역기[2, 3]를 통해 알 수 있듯이 제시된 작용식은 번역기를 쉽게 만들 수 있도록 보다 실제적이고, 구체적이고, 정확한 형식 의미로 명세된 사실을 확인할 수 있다.

6. 결 론

본 논문에서는 객체 지향 언어인 자바를 연산 의미론에 근거하여 정적이고 동적인 형식 의미를 명세하는 작용식을 제시하였다. 작용식은 명령형 언어뿐만 아니라 객체 지향 언어인 자바를 위한 정적이고 동적인 의미 구조를 정확하게 명세하고 있다.

본 논문에서 제시된 작용식은 연산 의미론에 근거하여 의미를 정의하여 기존의 표현법과 비교할 때 아주 간단하게 의미를 명세할 수 있고 번역기를 쉽게 구현할 수 있도록 보다 실제적이고 구체적이고 정확한 동적 의미구조를 명세하고 있다. 형식 의미를 명세할 때 8가지 속성과 13가지 모듈을 사용하고, 일반적인 제어 구문을 사용해 형식 의미를 표현하여 판독성과 모듈성을 향상시켰다. 객체를 처리하거나 예외를 처리하는 등의 언어 기능을 확장할 경우 작용식도 확장하고 추가하여 관련된 형식 의미를 정확하게 명세할 수 있다. 새로운 언어에 대한 형식 의미를 명세할 경우, 언어의

문법을 표현하는 작용식의 구문 안에 새롭게 명세할 언어의 명령문을 사용할 수 있고 8가지 속성을 사용하여 정적이고 동적인 의미를 얻을 수 있어 융통성 있는 형식 의미 명세법이다.

자바 언어에 대해 세 표기법인 DS, ASM, AS와 제시된 작용식(AE)을 판독성, 모듈성, 확장성, 융통성 영역에서 비교했을 때 모든 영역에서 작용식이 높이 평가되었고 제시된 작용식이 기존의 표현법보다 우수함을 확인할 수 있다.

본 논문에서 제시한 작용식에 정의된 의미 명세를 최적화 과정이나 번역과정에 사용하여 본 작용식이 구체적이고 실제적인 동적 의미구조를 표현하고 있음을 입증하는 연구가 향후 연구 과제로 진행되어야 할 것이다.

참 고 문 헌

[1] 한정란, “객체 지향 언어를 위한 의미 명세”, 인터넷정보학회논문지, 제8권 5호, pp.35-43, 2007.

[2] 한정란, 최성 “동적 의미 분석에 의한 점진 해석기 구축”, 인터넷정보학회논문지, 제5권 6호, pp.111-120, 2004.

[3] 한정란, 최성, “작용 식 기반 통합 점진 해석 시스템 구축”, 정보처리학회논문지, Vol.11-A, No.3, 2004.

[4] 한정란, JAVA 기초부터 활용까지, 21세기사, 2007.

[5] Jim Davies, David Faitelson and James Welch, Domain-specific Semantics and Data Refinement of Object Models, Electronic Notes in Theoretical Computer Science 195, pp.151-170, 2008.

[6] Ana Paula Ludtke Ferreira and Leila Ribeiro, A Graph-based Semantics For Object-oriented Programming Constructs, Electronic Notes in Theoretical Computer Science 122, pp.89-104, 2005.

[7] Yingzhou Zhang and Baowen Xu, “A Survey of Semantic Description Frameworks for Programming Languages,” ACM SIGPLAN Notices Vol.39(3), Mar, 2004.

[8] J. Alves-Foss, editor. Formal Syntax and Semantics of Java, Vol.1523 of Lecture Notes in Computer Science. Springer-Verlag.

[9] David A. Watt and Deryck F. Brown, “Formalising the Dynamic Semantics of Java,” 2006.

한 정 란

e-mail : jlhan@uhs.ac.kr

1985년 이화여자대학교 전자계산학과(학사)

1987년 이화여자대학교 컴퓨터공학과(이학석사)

1999년 이화여자대학교 컴퓨터공학과(공학박사)

1999년~현 재 협성대학교 경영정보학과 부교수

관심분야 : 형식언어론, 점진컴파일러, 전자상거래, XML 등