KIPS Trans. Comp. and Comm. Sys.
Vol.12, No.5 pp.157~164
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2023.12.5.157

Formal Semantics for Lambda Expression of Java

Han Jung Lan†

ABSTRACT

Specifying the semantic structure for functional interfaces and lambda expressions, which are the latest features of Java, can be referenced when designing similar functions in the future, and is also required in the process of standardizing or implementing an optimized translator. In this study, action equation 3.0 is newly proposed to express the static and dynamic semantic structure of functional interfaces and lambda expressions by modifying and upgrading the existing expressions to express the semantic structures of java functional interfaces and lambda expressions. Measure the execution time of java programs by implementing the semantic structure specified in action equation 3.0 in java, and prove that action equation 3.0 is a real semantic structure that can be implemented through simulation. The superiority of this action equation 3.0 is to be confirmed by comparing the action equation 3.0 specified in the four areas of readability, modularity, extensibility and flexibility with the existing representative semantic expression methods.

Keywords : Formal Semantics, Action Equation 3.0, Functional Interface, Specification of Semantics, Lambda Expression

자바 람다식에 대한 형식 의미론

한 정 란†

요 약

자바의 최신 기능인 함수 인터페이스와 람다식에 대한 의미구조를 명세하는 것은 향후 유사한 기능을 설계할 때 참고할 수 있고 표준화하는 과정이나 최적화된 번역기를 구현하는 과정에서도 필요하다. 본 연구에서는 자바의 함수 인터페이스와 람다식에 대한 의미구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 함수 인터페이스와 람다식의 정적·동적 의미구조를 표현하는 작용식 3.0을 새롭게 제시한다. 작용식 3.0에 명세된 의미구조를 자바로 구현해 자바 프로그램들에 대한 실행시간을 측정하고 시뮬레이션을 통해 작용식 3.0이 구현가능한 실제적인 의미구조 명세법인 것을 입증한다. 판독성(Readability) 측면, 모듈성(Modularity) 측면, 확장성(Extensibility) 측면, 융통성(Flexibility) 측면에서 대표적인 의미구조 표현법과 작용식 3.0을 비교하여 작용식 3.0의 우월성을 확인하고자 한다.

키워드 : 형식 의미론, 작용식 3.0, 함수 인터페이스, 의미구조 명세, 람다식

1. 서 론

프로그래밍 언어가 시대 흐름에 맞춰 진화하면서 자바에서 람다식(lambda expression)과 같은 최신 기능들이 새롭게 추가되고 이런 최신 기능을 다른 언어에도 접목하여 사용하는 것이 바람직하다. 언어의 서로 다른 기능들을 접목하기 위해 최신 기능의 의미구조를 정확하게 명세하는 연구가 필요하다. 그런데, 자바에 대한 형식 의미를 제시하는 연구가 많이 진행되었지만 간단한 자바 기능에 대해 의미구조를 명세하였고 자바의 최신 기능인 함수 인터페이스와 람다식에 대한 의미구조를 표현하는 연구는 이뤄지지 않았다.

현재까지 작용식과 작용식 2.0을 사용하여 자바의 예외처리와 파이썬의 의미구조를 명세하는 연구[1,2]가 진행됐지만, 자바의 최신 기능인 함수 인터페이스(functional interface)와 람다식에 대한 의미구조를 명세하는 연구가 없어서 기존의 작용식을 수정하고 업그레이드해서 함수 인터페이스와 람다식의 정적·동적 의미구조를 표현하는 연구가 필요하다고 사료된다. 기존 연구의 의미구조를 사용해서 자바의 최신 기능인 함수 인터페이스와 람다식 같은 기능을 명세할 수 없고, 번역기를 구현할 수 있도록 구체적이고 실제적인 동적 의미구조를 명세할 수 없어서 작용식 3.0을 제시해 함수 인터페이스와 람다식 같은 기능의 동적 의미구조를 명세한다.

본 연구에서는 자바의 함수 인터페이스와 람다식에 대한 의미구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 함수 인터페이스와 람다식의 정적·동적 의미구조를 표현하는 작용식 3.0을 새롭게 제시한다.

기존 연구에서는 프로그램 테스트를 통해 성능 평가가 이뤄지지 않았지만 본 연구는 구체적이고 실제적인 동적 의미

※ 이 연구는 2022년도 협성대학교 교내연구비 지원에 의한 연구임(20220047).
† 종신회원 : 협성대학교 컴퓨터공학과 교수
Manuscript Received : October 21, 2022
Accepted : December 1, 2022
* Corresponding Author : Han Jung Lan(jlhan@omail.uhs.ac.kr)

구조를 명세하여 예제 프로그램에 대해 자바로 구현하고 프로그램 실행시간과 시뮬레이션을 통해 성능을 평가하고자 한다.

작용식 3.0에서 정의된 의미 명세가 실제로 구현가능한 의미 명세인지 입증하기 위해 자바로 구현해 세 가지 유형의 프로그램의 실행시간을 측정하였고, 실행시간을 근거로 시뮬레이션을 통해 다양한 길이의 프로그램 실행시간을 추정함으로써 작용식 3.0의 동적 의미구조 명세를 통해 번역기를 구현할 수 있다는 사실을 입증한다.

본 연구에서 제시한 작용식 3.0의 의미구조를 분석하기 위해 판독성(Readability) 측면, 모듈성(Modularity) 측면, 확장성(Extensibility) 측면, 융통성(Flexibility) 측면에서 대표적인 의미구조 명세법과 비교하여 작용식 3.0이 우월함을 검증하고자 한다.

본 논문은, 2장에서는 다양한 의미구조를 명세하는 관련 연구에 대해 소개하고, 3장에서는 작용식 3.0을 제시하여 설명하고, 4장에서는 자바 프로그램 테스트를 통해 실행시간을 측정하고 시뮬레이션을 통해 작용식 3.0이 구현가능한 실제적인 동적 의미구조임을 입증하고, 기존 연구에서 사용한 의미구조 분석을 통해 작용식 3.0과 비교해 평가하고, 5장에서 결론을 기술한다.

2. 관련 연구

국내 연구 동향을 살펴보면 작용식 2.0을 사용해 파이썬에 대한 의미구조 명세[1] 연구가 진행되었고, 자바 언어의 예외처리에 대한 형식 의미론을 제시한 연구[2]가 진행되었고, 이 연구는 자바 람다식을 사용하기 전의 의미 명세이고 람다식과 관련한 연구는 없었다. 외국 연구의 경우 구현과 관련해 프로그래밍 언어 자체에 대한 의미를 명세하는 연구가 진행되었다. 파이썬의 서브셋 minpy에 대한 연산적 의미(operational semantics) 구조 연구[6]가 있고, minpy에서 확장된 기능에 대해 명세하는 파이썬 3.3의 연산적 의미 연구[7]가 있다.

파이썬에 대한 의미구조 연구[1]에서는 기존의 작용식을 수정하고 업그레이드해서 파이썬의 정적이고 동적인 의미구조를 표현하는 작용식 2.0을 새롭게 제시하여 파이썬의 의미구조를 명세하였다[1].

파이썬의 서브셋 minpy에 대한 연산적 의미 연구[6]에서는 프로그래밍 언어 중 하나인 Haskell을 사용해 연산적 의미를 명세하고 있다[6]. 추상 기계(abstract machine)의 상태 전이에 의해 의미구조를 정의하고, 상태 전이는 재작성 규칙으로 정의해 시스템 상태를 변환하고 있다[6].

파이썬 3.3에 대한 연산적 의미 연구[7]에서는 해석 도구를 사용해 프로그램의 상태 공간(state space)을 탐색하고 프로그램에 대한 정적 추론(static reasoning)을 수행하며, 파이썬에 대한 연산적 의미를 명세하기 위해 해석 도구를 제공하는 K 의미구조(K Semantic Framework)를 사용한다[7].

파이썬을 사용한 구조적 실행 의미구조 연구[8]에서는 방문자 패턴과 예외를 처리하는 기능을 이용하여 ML 스타일의 간단한 함수형 언어에 대해 구조적 실행 의미구조를 구현하는 기법을 소개하고, 학습 난이도가 높고 비교적 덜 알려진 Haskell, ML, Scheme 등과 같은 전통적인 함수형 언어를 사용하지 않고, 파이썬을 사용하여 프로그래밍 언어의 핵심 개념과 관련된 구현 기법을 설명하고 있다[8].

자바에 대한 의미 표현을 정의한 연구들[9-11]을 살펴보면 크게 세 가지 표현법인 Denotational Semantics(DS) 표현, Abstract State Machine(ASM) 표현, Action Semantics (AS) 표현으로 분류할 수 있다[2].

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[2,9].

ASM(Abstract State Machine) 표현은 자바의 전체 구문으로 확장 가능하지만 많은 구조를 처리하기에는 불완전한 측면이 있다[2,9]. ASM은 자바의 실제 구문과는 다른 추상구문으로 매우 명확하지만 저급 수준이고, 제한된 의미에서만 모듈방식으로 사용된다[2,9].

AS(Action Semantics) 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[2,9].

자바의 최신 기능인 함수 인터페이스와 람다식에 대한 의미구조를 표현하는 연구가 없어서 본 연구에서는 함수 인터페이스와 람다식에 대한 의미구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 실제적으로 번역기를 구현할 수 있도록 람다식과 함수 인터페이스의 정적·동적 의미구조를 표현하는 작용식 3.0을 제시한다. 함수 인터페이스와 람다식에 대해 정적·동적 의미구조를 표현하면서, 이전 연구에는 없는 람다식과 인터페이스의 메서드들의 의미구조를 구체적으로 표현함으로써 다른 자바 연구와 비교해 볼 때 자바의 최신 기능을 동적 의미 구조로 명세할 수 있도록 손쉽게 확장할 수 있음을 프로그램 테스트를 통해 보여준다.

본 연구에서는 프로그래밍 언어의 융통성을 높이고 작용식 3.0을 사용하여 번역기를 보다 쉽게 구현할 수 있도록 함수 인터페이스와 람다식에 대한 정적·동적 의미구조를 명세하고자 한다.

3. 람다식과 작용식 3.0

람다식(lambda expression)은 메서드를 하나의 식으로 표현한 것으로 이름이 없는 메서드이고, Java 8부터 람다식을 사용할 수 있다. 함수 인터페이스는 상수(멤버 변수) 없이

하나의 메서드만 선언한 인터페이스이다. Java 8부터 함수 인터페이스를 사용하고, 함수 인터페이스의 경우, 인터페이스의 메서드를 구현하는 클래스를 만들지 않더라도 함수 인터페이스 객체를 생성하면서 람다식으로 메서드를 작성한 후 메서드를 호출하여 사용할 수 있다. 이전 버전의 자바에서는 인터페이스로 객체를 바로 생성할 수 없었는데, 함수 인터페이스의 경우 람다식을 정의하여 인터페이스 타입의 객체를 생성할 수 있다. 람다식을 사용하면 불필요한 코드를 줄여주어 간결하고 작성된 코드의 가독성을 높여줄 수 있다.

자바에서는 람다식을 ( ) -> { }으로 나타내어 작성한다. ( ) 안에 람다식에서 사용할 매개변수들을 쉼표(,)로 분리하여 자료형과 같이 선언하고, { } 안에 처리할 명령문들을 기술한다. 매개변수의 자료형을 유추할 수 있을 때 자료형은 생략가능하다. 그러나, 두 개 이상의 메서드를 선언한 인터페이스에서는 람다식을 사용할 수 없다. 함수 인터페이스와 람다식을 처리하기 위해 기존의 작용식을 확장하고 수정해 작용식 3.0을 사용해서 의미구조를 명세한다.

람다식 형식
(매개변수목록) -> { 문장 }
람다식 사용 예
```
(int x, int y)->{ return x+y; } // x와 y의 합을 반환
```

함수 인터페이스로 선언한 메서드를 호출해 사용하기 위해, 먼저 함수 인터페이스 타입의 객체를 선언하고 람다식을 작성해 정의하면 메서드를 호출할 수 있다. 특히, 기존의 인터페이스를 사용하는 방법처럼 인터페이스를 구현하는 클래스를 작성하지 않아도 람다식을 활용하면 함수 인터페이스의 메서드를 람다식으로 정의한 후 메서드를 바로 호출할 수 있다.

함수 인터페이스를 작성한 후, 함수 인터페이스 객체를 선언하고 람다식을 사용해 메서드를 작성하는 방법은 다음과 같다.

함수 인터페이스 작성 형식
```
@FunctionalInterface      // 생략 가능
interface 함수인터페이스이름 {
   // 메소드 원형
}
```
함수 인터페이스 사용 예
```
@FunctionalInterface      // 생략 가능
interface FunctionInterface {
   public void methodFunc();
}
```
함수 인터페이스 객체 선언과 호출 형식
```
함수인터페이스이름 객체이름;
객체이름 = 람다식;
// 함수인터페이스이름 객체이름 = 람다식;
...
객체이름.메서드이름()          // 메서드 호출
```
함수 인터페이스 객체 선언과 호출 예1
```
FunctionInterface fi; // 함수인터페이스객체 선언
fi = () -> { String str = "람다식 사용";
             System.out.println(str);     };
 // "람다식 사용"을 출력하는 메서드
...
fi.methodFunc()           // 메서드 호출
```
함수 인터페이스 객체 선언과 호출 예2
```
FunctionInterface fi=()->{ String str="람다식 사용";
                           System.out.println(str);
};
...
fi.methodFunc()           // 메서드 호출
```

3.1 인터페이스

자바는 인터페이스를 제공하여 자바의 다형성을 극대화하고 개발하는 코드를 수정하는 것을 줄이고 프로그램의 유지 보수성을 높인다. 작용식 3.0에서 make_table, make_table2, lookup, search_member 네 가지 모듈을 사용해 인터페이스를 처리하는 의미구조를 명세한다. 인터페이스와 관련된 정보를 심벌테이블에 저장하기 위해 make_table 모듈을 사용하고, make_table2 모듈은 인터페이스 객체와 관련된 정보를 저장하기 위한 것이고, 인터페이스 메서드의 매개변수를 정의할 때 선언된 자료형을 찾기 위해 lookup 모듈을 사용하고, search_member 모듈은 인터페이스 이름이 주어졌을 때 그 인터페이스 안에 선언된 멤버 변수들을 반환하는 모듈이다. searchIntName 모듈은 인터페이스 객체이름이 주어졌을 때 인터페이스 이름을 반환하는 모듈이다.

특히, 함수 인터페이스는 상수(멤버 변수) 없이 하나의 메서드만 제공하는 것이고 람다식을 사용해 메서드를 간단하게 정의하고 처리할 수 있다. interface_id.type는 상수 없이 메서드 하나만 정의한 인터페이스의 경우 메서드를 람다식으로 처리하기 위해 함수 인터페이스인지 타입을 설정하는 것이다. determine_fun 모듈은 상수 없이 하나의 메서드만 선언한 인터페이스인지 판별해서 interface_id.type을 결정하는 모듈이고, 함수 인터페이스인 경우 interface_id.type은 “function”이고 일반 인터페이스일 경우 “plain“이다. determine_env 모듈은 식별자(ID) 이름이 주어졌을 때 식별자의 선언 환경을 반환하는 모듈이다.

작용식 2.0에서처럼 out은 작용식 3.0을 실행한 후에 발생하는 결과를 나타내는 속성이고, val 속성은 각 비단말

(nonterminal)의 값을 나타내는 속성이고, name 속성은 식별자의 이름을 나타내는 속성이다[1]. env 속성은 어떤 클래스나 인터페이스에 속한 것인지 나타내는 속성이고, penv 속성은 슈퍼 클래스나 슈퍼 인터페이스가 뭔지 해당 슈퍼 클래스나 슈퍼 인터페이스를 나타내는 속성이다.

◀Execute [statements]→**event**〖complete|diverge〗
- Execute [<$stmt_1$>,<$stmt_2$>,...<$stmt_n$> where n ≥1] →
  $stmt_1$.out ← Execute [<$stmt_1$>]
  $stmt_2$.out ← Execute [<$stmt_2$>]
  ...
  $stmt_n$.out ← Execute [<$stmt_2$>]

인터페이스에 대한 의미구조를 표현하는 작용식 3.0에서 함수 인터페이스와 일반 인터페이스에 대한 의미구조를 명세하고, 함수 인터페이스일 경우 람다식으로 메서드를 정의하고, 일반 인터페이스일 경우 클래스를 정의하면서 메서드를 구현한다.

본 연구에서 함수 인터페이스와 람다식을 위한 의미 구조를 명세할 때 다음 세 단계의 작업에 대해 의미구조를 명세한다.

1. 상수 없이 하나의 메서드(추상 메서드)만 갖는 함수 인터페이스의 작성
2. 함수 인터페이스를 생성하는 객체 선언문을 통해 객체를 생성하면서 메서드를 람다식으로 작성
3. 작성된 람다식 호출

<interface 문의 Execute equation>

- Execute [**public interface** <interface_id>] →
  interface_id.env ← **public**
  make_table(interface_id.name, interface_id.env)
  interface_id.addr← current_point
  interface_id.type←determine_fun(interface_name)
- Execute [**interface** <interface_name>] →
  interface_id.env ← interface_id.name
  make_table(interface_id.name, interface_id.env)
  interface_id.addr← current_point
  interface_id.type← determine_fun(interface_name)
- Execute [**interface** <interface_id> **extends** <derived_inter_id>]→
  interface_id.env ← interface_id.name
  interface_id.penv ← derived_inter_id.name
  make_table(interface_id.name, interface_id.env)
  interface_id.addr ← current_point
- Execute [**public interface** <interface_id> **extends** <derived_inter_id>]→
  interface_id.env ← **public**
  interface_id.penv ← derived_inter_id.name
  make_table(interface_id.name, interface_id.env)
  interface_id.addr ← current_point
- Execute [<modifier><id>]→
  id.name ← id
  environment.name ← determine_env(id.name)
  id.env ← environment.name
  make_table(id.name, id.env)
- Execute [<modifier><id_1>, <id_2>, <id_n> where n≥1]→
  for i=1 to n do
  id_i.name ← id_i
  environment.name ← determine_env(id_i.name)
  id_i.env ← environment.name
  make_table(id_i.name, id_i.env)
  od
- Execute [**public class** <class_id> **implements** <derived_interface>]→
  class_id.env ← **public**
  class_id.penv ← derived_interface.name
  make_table(class_id.name, class_id.env)
  class_id.addr ← current_point
- Execute[<obj_id>**= new** <class_id>(<par_1>,..,<par_n>) where n≥)]→
  obj_id.env ← class_id.name
  make_table(obj_id.name, obj_id.env)
  member_id ← search_member(class_id.name)
  for each member_id do
  make_table(member_id.name, obj_id.name)
  do
  obj_id.addr ← current_point
  for i=1 to n do
  $param_i$.env ← class_id.env
  $param_i$.val ← Eval_out[<$param_i$>]
  od
  return_save(class_id.addr)
  control_transfer(class_id.addr)
- Execute [<class_id><obj_id> **= new** <class_id>(<par_1>, ..., <par_n>) where n ≥1)]→
  obj_id.env ← class_id.name
  make_table(obj_id.name, obj_id.env)
  member_id ← search_member(class_id.name)
  for each member_id do
  make_table(member_id.name, obj_id.name)
  do
  obj_id.addr ← current_point
  for i=1 to n do
  $param_i$.env ← class_id.env
  $param_i$.val ← Eval_out[<$param_i$>]
  od
  return_save(class_id.addr)
  control_transfer(class_id.addr)

다음 예제는 함수 인터페이스를 사용해 1부터 n까지 짝수와 홀수의 합을 구하는 프로그램으로 함수 인터페이스 LambdaCompute를 선언하고 n을 입력받고 1부터 n까지 짝수의 합과 홀수의 합을 계산하여 짝수의 합은 even에, 홀수의 합은 odd에 저장하여 각각의 합을 출력하는 compute() 메서드를 함수 인터페이스 객체를 생성하면서 람다식으로 작성하는 프로그램이다. 함수 인터페이스의 메서드를 호출할 때 "함수인터페이스객체이름.메서드이름(인수)" 즉, "la.compute(n)"으로 호출한다. 람다식으로 compute() 메서드를 작성할 때 매개변수가 하나일 때 ( )는 생략가능하고 매개변수 타입을 유추할 수 있으면 타입도 생략가능해서 "num -> { }"처럼 람다식을 작성한다.

함수 인터페이스의 경우 메서드를 람다식으로 작성하기 전에 함수 인터페이스인지 체크해야 한다. 함수 인터페이스만 람다식을 사용할 수 있고 함수 인터페이스의 interface_id.type은 "function"이다.

<함수 인터페이스 객체 선언문의 Execute equation>

• Execute[<interface_id><obj_id_1>,<obj_id_2>,...,<obj_id_n>; where n ≥1]→
  if **interface_id.type** is function then
    for i=1 to n do
      obj_id_i.env ← interface_id.name
      make_table(obj_id_i, obj.env)
    od
  endif

```
import java.util.Scanner;
interface LambdaCompute{  void compute(int n);  }
public class LamSumEx {
 public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  System.out.print("1 ~ n odd & even sum program);
  System.out.print(" input n : ");
  int n=scanner.nextInt();
  if (n<0) n=-n;
  LambdaCompute la= num -> {
    int odd=0, even=0;
    for(int i=0; i<=num; i++)  // calculate sum
       if (i%2==0) even+=i;
       else odd+=i;
    System.out.print("sum of even : "+even);
    System.out.println(", sum of odd : "+odd);
   };
 la.compute(n);
 scanner.close(); }
}
```

<함수 인터페이스 객체 생성문의 Execute equation>

• Execute[<obj_id> = [(<par_0>, <par_1>, ... ,<par_n>) -> {<lambda_st$_n$>, ... ,lambda_st$_n$>} where n≥0]→
 if **interface_id.type** is function then
  interface_id.name←searchIntName(obj_id.name)
  obj_id.env ← interface_id.name
  mem_id ← search_member(interface_id.name)
  mem_id.type ← lookup(method.env,method.name)
  for each mem_id do
   make_table2(mem_id.name, mem_id.type, obj_id.name)
  od
  for i=0 to n do
   par_i.env ← interface_id.env
   par_i.val ← Eval_out[<par_i>]
  od
  return_save(interface_id.addr)
  control_transfer(interface_id.addr)
endif
• Execute[<interface_id><obj_id>= [(<par_0>,...,<par_n>) -> { <lambda_st> } where n≥0]→
 if **interface_id.type** is function then
  obj_id.env ← interface_id.name
  make_table(obj_id.name, obj_id.env)
  mem_id ← search_member(interface_id.name)
  mem_id.type ← lookup(method.env,method.name)
  for each mem_id do
   make_table2(mem_id.name,mem_id.type,obj_id.name)
  od
  for i=0 to n do
    par_i.env ← interface_id.env
    par_i.val ← Eval_out[<par_i>]
  od
  return_save(interface_id.addr)
  control_transfer(interface_id.addr)
 endif
• Execute [<lambda_st$_1$>,<lambda_st$_2$>,... <lambda_st$_n$> where n ≥1] →
  lambda_st$_1$.out ← Execute [<lambda_st$_1$>]
  lambda_st$_2$.out ← Execute [<lambda_st$_2$>]
  ...
  lambda_st$_n$.out ← Execute [<lambda_st$_n$>]

4. 프로그램 테스트 및 작용식 3.0의 의미구조 비교 평가

의미구조를 명세하는 대표적인 세 표기법인 DS(Denotational Semantics), ASM(Abstract State Machine), AS(Action Semantics)와 작용식 3.0을 비교해서 작용식 3.0의

의미구조의 우월성을 확인한다. 작용식 3.0의 경우, interface_id.type을 사용해 상수(멤버변수)없이 메서드 하나만 정의한 인터페이스의 경우 메서드를 람다식으로 작성할 수 있다. 작용식 3.0의 경우, 실행하는 의미구조를 쉽게 표현하고 함수 인터페이스일 경우 람다식으로 처리하도록 구체적으로 명세하고 있어 작용식 3.0이 판독성이 뛰어남을 확인할 수 있다.

Table 1. Execution Time of Java Program (unit: us)

Program Type	Assign. Stmt.	Condition Stmt.	Loop	lambda expression
M1	0.5	1	2	5
M2	0.5	1	2	5
P1	0.5	1	2	4.5
P2	0.5	1	2	4.5
S1	0.5	1	2	4.1
S2	0.5	1	2	4.1

작용식 3.0에서 정의된 의미구조가 실제로 구현가능한 의미구조 명세법인지 입증하기 위해, 작용식 3.0의 의미구조를 자바로 구현해 세 가지 유형의 람다식을 사용한 프로그램으로 실행시간을 측정해서 Table 1에 나타내었다.

Table 1에서 M1, M2는 함수 인터페이스와 람다식을 사용해 최댓값과 최솟값과 합과 평균을 계산하는 프로그램이고, P1, P2는 함수 인터페이스와 람다식을 사용해 급료를 구하는 프로그램이고, S1, S2는 함수 인터페이스를 사용해 합계를 계산하는 람다식을 호출하는 프로그램이다. 각 유형의 프로그램에 대해 배정문(Assignment statement), 조건문(Conditional statement), 반복문(Loop), 람다식(lambda expression)에 대한 실행시간을 측정하였다. 프로그램 테스트를 통해 알 수 있듯이 람다식을 활용한 예제에 대해서 작용식 3.0에서 명세한 의미 구조를 자바로 구현함으로써 작용식 3.0이 기존의 국외 자바 의미 구조와는 달리 번역기를 구현할 수 있는 구체적이고 실제적인 의미 구조라는 사실을 확인할 수 있다.

Table 1의 프로그램 실행시간을 근거로 시뮬레이션을 통해 Table 2에서 다양한 길이를 갖는 위의 세 가지 유형의 프로그램의 실행시간 추정치를 표시하였다.

실행 환경과 조건이 달라서 다른 의미구조와 성능 측면에서 객관적으로 서로 비교할 수 없고, 기존 국외 자바 연구에서 성능 테스트를 실행하지 않아 기존 연구와는 비교할 수 없지만, 자바의 최신 기능인 함수 인터페이스와 람다식을 명세한 작용식 3.0 의미구조가 번역기를 구현할 수 있는 동적 의미구조를 실제적으로 명세하고 있음을 프로그램 테스트를 통해 확인할 수 있다. 본 작용식 3.0은 형식적인 의미구조가 아니라 보다 실제적이고 구체적인 동적 의미구조이고, 작용식 3.0 의미구조를 번역기로 구현할 수 있다. 따라서, 작용식 3.0은 자바 람다식과 같은 최신 기능을 명세할 수 있는 융통성 있는 동적 의미 구조임을 확증하고 있다.

Table 2. Execution Time for No. of Program Lines (unit: us)

Program Type	No. of program lines				
	50	500	1000	5000	10000
P1	495	4953	9903	49513	99022
P2	490	4950	9895	49471	98940
P3	481	4815	9627	48132	96263

의미구조를 비교한 국내외 연구에서는, 명세된 의미구조의 성능을 판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility) 측면에서 평가하고 있다[1]. 작용식 3.0은 기존 작용식을 확장해 함수 인터페이스와 자바 람다식의 의미구조를 명세한 것이고, 자바 람다식에 대한 의미구조를 비교 분석한 관련 연구가 없으므로 객체지향 언어인 자바에 대한 의미구조를 분석한 기존 국외 연구[9-11]들과 비교함으로써 작용식 3.0의 효율성을 확인하려고 한다.

기존 연구[9-11]들을 중심으로 Table 3에 세 표기법인 DS, ASM, AS을 자바 언어에 대한 판독성 측면, 모듈성 측면, 확장성 측면, 융통성 측면에서 작용식(AE) 3.0과 비교한 결과를 표시하였다. Table 3에서 세 표기법 DS, ASM, AS에 대한 평가는 기존 국외 자바 논문[9-11]에서 분석해 판정한 내용을 그대로 나타내었다.

먼저, 판독성 측면에서, ASM이나 AS와는 달리 작용식 3.0의 경우 if나 for같은 제어 구문을 사용하여 자바 람다식의 동적 의미구조를 정확하고 구체적으로 명세한다. 특히 자바의 최신 기능인 함수 인터페이스와 자바 람다식의 동적 의미구조를 쉽게 판독할 수 있어 판독성이 뛰어남을 알 수 있다.

모듈성 측면에서 모듈성이 높은 AS와 비교할 때, 작용식 3.0의 경우 기존 연구에서 자바와 간단한 객체지향 기능 및 파이썬에 대한 형식 의미구조를 명세하는 작용식[1-3]을 확장하여 함수 인터페이스와 자바 람다식의 동적 의미구조를 명세하기 위해 새로운 모듈을 추가하여 동적 의미구조를 구체적으로 명세하고 있고, 작용식 3.0으로 명세된 각각의 기능을 수행하는 모듈을 실제로 작성하여 번역 과정을 수행함으로 모듈성이 뛰어남을 알 수 있다.

Table 3. Comparison of Formal Semantics

Semantics	Readability	Modularity	Extensibility	Flexibility
DS	low	low	low	medium
ASM	medium	medium	high	high
AS	medium	high	high	low
AE 3.0	high	high	high	high

확장성 측면에서 작용식 3.0을 ASM이나 AS와 비교할 때, 작용식 3.0의 경우, 자바의 최신 기능인 함수 인터페이스와 자바 람다식의 동적 의미구조를 표현하여 명세할 언어 기능이 시대 흐름에 맞춰 확장되더라도 새로운 동적 의미구조를 정확하게 명세할 수 있다. 작용식 3.0은 작용식을 기반으로 함수 인터페이스와 자바 람다식의 동적 의미구조를 명세하기 위해 기능을 확장한 것이므로 자바의 최신 기능도 명세할 수 있는 확장 가능성이 뛰어난 동적 의미구조 명세 방법이다.

ASM과 융통성 측면에서 비교할 때, 간단한 OBLA 언어[5]와 예외처리 기능이 추가된 객체 지향 언어 구문[2]과 파이썬의 의미구조 명세[1]뿐만 아니라, 작용식 3.0에서 문법 구문만 변경하여 함수 인터페이스와 자바 람다식의 동적 의미구조를 정확하게 명세할 수 있어 융통성이 뛰어난 의미 명세 방법이라는 사실을 알 수 있다. 작용식 3.0을 사용하여 자바의 최신 기능인 함수 인터페이스와 람다식의 동적 의미구조를 구현하는 과정을 더 효율적으로 수행할 수 있다.

연산 의미론(operational semantics)을 근거로 작용 식 3.0은 번역기를 구현할 수 있도록 정확하고 실제적인 의미구조를 제시하였고 무엇보다 정적 의미구조뿐만 아니라 동적 의미구조도 정확하게 명세하고 있다. 작용식 3.0을 사용해 빠르고 쉽게 번역기를 구현할 수 있도록 동적 의미구조를 명세하고 있고, 번역기를 구현하는 과정에 작용식 3.0을 활용할 수 있어, 보다 정확하고, 실제적이고, 구체적인 동적 의미구조를 명세하는 것이 가능함을 자바 예제 프로그램들의 실행시간 테스트를 통해 확인할 수 있다. 따라서, 최적화나 번역기를 구현하는 과정에 형식 의미를 정확하게 명세한 작용식 3.0을 사용함으로써 번역기의 구현 효율성을 증대할 수 있다.

5. 결 론

본 논문에서는 연산 의미론에 근거하여 자바의 최신 기능인 함수 인터페이스와 람다식의 정적·동적 형식 의미구조를 명세하는 작용식 3.0을 제시하였다. 작용식 3.0은 작용식을 기반으로 자바의 함수 인터페이스와 자바 람다식의 의미구조 명세를 위해, 의미구조 명세 기능을 확장한 것으로 자바의 최신 기능인 함수 인터페이스와 람다식에 대한 정적·동적 의미구조를 명세하는 방법이다.

본 논문에서 제시된 작용식 3.0은 연산 의미론에 근거하여 형식 의미구조를 정의하여 기존의 의미구조 표현 방법과 비교할 때 간단하고 쉽게 동적 의미구조를 명세할 수 있고, 명세된 의미구조를 사용해 번역기를 빠르게 구현할 수 있도록, 보다 정확하고 실제적이고 구체적인 형식 명세 방법으로 동적 의미구조를 제시하고 있다.

작용식 3.0에서 명세한 의미구조를 세 가지 유형의 프로그램에 대해 자바로 구현해 실행시간을 측정하였고 프로그램의 실행시간을 근거로 다양한 길이의 프로그램에 대한 시뮬레이션을 통해 실행시간 추정치를 계산함으로써 작용식 3.0의 의미구조 표현법을 사용해 번역기를 실제로 구현할 수 있음을 확인할 수 있다.

대표적인 의미구조 표기법인 DS, ASM, AS와 본 논문에서 제시한 작용식 3.0을 판독성 측면, 모듈성 측면, 확장성 측면, 융통성 측면에서 비교 평가했을 때 모든 측면에서 작용식 3.0이 높이 평가되었고 기존의 대표적인 의미구조 명세법보다 작용식 3.0이 우수함을 확인할 수 있다.

References

[1] J. L. Han, “Formal semantics based on action equation 2.0 for python,” *KIPS Transactions on Computer and Communication Systems*, Vol.10, No.6, pp.163-172, 2021.

[2] J. L. Han, “Formal semantics for processing exceptions,” *KIPS Transactions on Computer and Communication Systems*, Vol.17, No.4, pp.173-180, Apr. 2010.

[3] J. L. Han, “Specification of semantics for object oriented programming language,” *KSII Transactions on Internet and Information Systems*, Vol.8, No.5, pp.35-43, 2007.

[4] J. L. Han and Sung Choi, “Building of integrated increment interpretation system based on action equations,” *The KIPS Transactions: Part A*, Vol.11, No.3, pp.149-156, 2004.

[5] J. L. Han, “Incremental interpretation based on action equations,” Ph. D Thesis Ewha Womans University, 1999.

[6] G. J. Smeding, “An executable operational semantics for Python,” Master’s thesis, Utrecht University, 2009.

[7] D. Guth, “A formal semantics of Python 3.3,” Master’s thesis, University of Illinois, 2013.

[8] S. Ji and H. Im, “Implementing structural operational semantics in Python,” *Jorunal of KIISE*, Vol.45, No.11, pp.1176-1184, 2018.

[9] Y. Zhang and B. Xu, “A survey of semantic description frameworks for programming languages,” *ACM SIGPLAN Notices*, Vol.39, No.3, pp.14-30, 2004.

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

[11] D. A. Watt and D. F. Brown, “Formalising the dynamic semantics of Java,” 2006.

한 정 란

https://orcid.org/0000-0002-2770-6762
e-mail : jlhan@omail.uhs.ac.kr
1985년 이화여자대학교 전자계산학과 (학사)
1987년 이화여자대학교 컴퓨터학과(석사)
1999년 이화여자대학교 컴퓨터공학과 (박사)
1999년 ~ 현 재 협성대학교 컴퓨터공학과 교수
관심분야 : Formal Semantics & Block Chain Application