KIPS Trans. Comp. and Comm. Sys.
Vol.10, No.6 pp.163~172
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2021.10.6.163

Formal Semantics Based on Action Equation 2.0 for Python

Jung Lan Han†

ABSTRACT

To specify a formal semantics for a programming language is to do a significant part for design, standardization and translation of it. The Python is popular and powerful, it is necessary to do research for a formal semantics to specify a static and dynamic semantics for Python clearly in order to design a similar language and do an efficient translation. This paper presents the Action Equation 2.0 that specifies a formal semantics for Python to change and update Action Equation. To measure the execution time for Python programs, we implemented the semantic structure specified in Action Equation 2.0 in Java, and prove through simulation that Action Equation 2.0 is a real semantic structure that can be implemented. The specified Action Equation 2.0 is compared to other descriptions, in terms of readability, modularity, extensibility, and flexibility and then we verified that Action Equation 2.0 is superior to other formal semantics.

Keywords : Formal Semantics, Action Equation 2.0, Specification of Semantics, Python

작용식 2.0 기반 파이썬에 대한 형식 의미론

한 정 란†

요 약

프로그래밍 언어의 형식적인 의미를 적절하게 표현하면 언어를 표준화하고 최적화하여 번역하는 과정에서 중요한 역할을 수행한다. 파이썬은 주목받는 강력한 언어이고, 파이썬에 대한 형식적인 의미 구조를 정의하고 표현하는 것은 향후 유사한 언어를 설계할 때 참고할 수 있고 표준화하는 과정이나 최적화된 번역기를 구현하는 과정에서도 필요하다. 본 연구에서는 파이썬에 대한 의미 구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 파이썬의 정적이고 동적인 의미 구조를 표현하는 작용식 2.0을 새롭게 제시한다. 작용식 2.0에 명세된 의미구조를 자바로 구현해 파이썬 프로그램들에 대한 실행시간을 측정하고 시뮬레이션을 통해 작용식 2.0이 구현 가능한 실제적인 의미 구조임을 입증하고, 판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility)의 네 영역에서 명세된 작용식 2.0을 기존의 대표적인 의미 표현법과 비교하여 본 작용식 2.0의 우월성을 확인하고자 한다.

키워드 : 형식 의미론, 작용식 2.0, 의미구조 명세, 파이썬

1. 서 론

파이썬은 코드 판독성(Readability)이 뛰어나고 간결하게 코딩을 할 수 있는 범용 프로그래밍 언어로 인터프리팅 방식으로 실행된다. 파이썬이 각광받는 이유는 파이썬으로 코딩하면 프로그램의 생산성이 뛰어나고 간결하면서도 효율적이고 빠르게 프로그램을 작성할 수 있기 때문이다.

어떤 언어를 설계하려면 그 언어에 대한 형식적인 의미를 파악해야 하고 형식적인 의미를 적절하게 표현하면 그 언어를 표준화하고 최적화하여 번역하는 과정에서 중요한 역할을 수행할 수 있다. 파이썬이 주목받으면서 널리 사용되는 언어이므로 파이썬에 대한 형식적인 의미 구조를 정의하고 표현하는 것은 향후 유사한 언어를 설계할 때 참고할 수 있고 표준화하는 과정이나 최적화된 번역기를 구현하는 과정에서도 필요한 연구라고 사료된다.

파이썬은 다른 언어에 없는 독특한 기능을 갖고 있는데 이러한 파이썬의 의미 구조를 구체적으로 명세하는 연구가 필요하다. 파이썬의 독특한 의미 구조를 구체적으로 명세하면 파이썬에서 명세된 기능을 다른 언어에도 새롭게 추가할 수 있고 표준화 작업을 수행할 때도 적절하게 활용할 수 있다.

기존에 작용식을 사용해 자바와 같은 언어에 대한 형식 의미를 제시하는 연구가 진행되었다. 그런데 파이썬의 경우 들여쓰기로 블록을 설정하고, list, tuple, set, dictionary 같은 다양한 컬렉션을 사용하고 있고, 함수 호출과 관련된 여러

※ 이 연구는 2020년도 협성대학교 교내연구비 지원에 의한 연구임(20200041).
† 종신회원 : 협성대학교 컴퓨터공학과 교수
Manuscript Received : December 15, 2020
Accepted : January 6, 2021
* Corresponding Author : Jung Lan Han(jlhan@omail.uhs.ac.kr)

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/ licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

가지 독특한 기능이 있어서 기존의 작용식으로는 의미 구조를 제대로 표현하지 못하는 한계점이 있다. 이러한 독특한 기능을 가진 파이썬의 의미 구조를 효과적으로 표현하는 것은 언어의 표준화나 올바른 번역을 위해 의미 있는 연구 분야라 할 수 있다.

본 연구에서는 파이썬에 대한 의미 구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 파이썬의 정적이고 동적인 의미 구조를 표현하는 작용식 2.0을 새롭게 제시한다.

작용식 2.0에서 정의된 의미 구조가 실제로 구현가능한 의미 구조인지 입증하기 위해 자바로 구현해 네 가지 유형의 프로그램의 실행시간을 측정한 값으로 시뮬레이션을 통해 다양한 길이의 프로그램의 대한 실행시간 추정치를 계산함으로써 작용식 2.0에 명세된 의미 구조로 번역기를 구현할 수 있다는 사실을 입증한다.

판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility)의 네 영역에서 명세된 작용식 2.0을 기존의 대표적인 의미 표현법과 비교하여 본 작용식 2.0의 우월성을 확인하고자 한다.

본 논문의 구성을 살펴보면 2장에서는 의미 구조를 표현하는 관련 연구에 대해 소개하고, 3장에서는 작용식 2.0을 제시하여 설명하고, 4장에서는 기존 연구들의 의미 구조 표현법을 명세하여 실제적인 의미 구조를 서로 비교하고, 시뮬레이션을 통해 작용식 2.0이 구현가능한 실제적인 의미 구조임을 입증하고, 5장에서는 기존 국내외 연구에서의 의미 구조 분석 방법을 사용하여 작용식 2.0을 평가하고, 6장에서 결론을 기술한다.

2. 관련 연구

국내의 경우 작용식을 사용해 객체 지향 특성을 처리하는 자바 언어에 대한 형식 의미론을 제시한 연구[1]가 진행되었고 외국의 경우 구현과 관련된 언어 자체에 대한 의미론을 명세하는 연구가 진행되었다. 파이썬의 서브셋 minpy에 대해 연산적 의미(Operational semantics) 구조를 명세한 연구[5]가 있고, minpy에서 확장된 기능을 중심으로 파이썬 3.3의 연산적 의미를 명세한 연구[6]가 있다.

파이썬의 서브셋 minpy에 대한 연산적 의미 연구[5]에서는 Haskell이라는 프로그래밍 언어로 연산적 의미를 명세하고 있다. 파이썬을 위한 연산적 의미는 추상 기계(Abstract machine)의 상태 전이에 의해 정의하고 상태 전이는 시스템 상태를 변환하는 재작성 규칙으로 정의하고 있다[5].

파이썬 3.3에 대한 연산적 의미 연구[6]에서는 프로그램의 상태 공간(State space)을 탐색하고 프로그램에 대한 정적 추론(Static reasoning)을 수행하기 위해 해석 도구를 사용하고 그러한 해석 도구를 제공하는 K 의미 구조(K Semantic Framework)에서 파이썬에 대한 연산적 의미를 명세한다[6].

파이썬을 사용한 구조적 실행 의미 구조 연구[7]에서는 ML 스타일의 간단한 함수형 언어에 대해 파이썬에서 방문자 패턴과 예외 처리를 이용하여 작은 보폭으로 정의된 구조적 실행 의미구조를 구현하는 기법을 소개하고, 학습 난이도가 높고 비교적 덜 알려진 ML, Haskell, Scheme 등과 같은 전통적인 함수형 언어 대신에, 파이썬을 활용하여 프로그래밍 언어 이론의 핵심 개념과 관련 구현 기법을 설명하고 있다.

객체 지향 언어의 의미를 정의한 논문들 중에서 자바에 대한 의미 표현을 정의한 연구들[8-10]을 살펴보면 크게 세 가지 표현법 Alves-Foss와 Lam의 Denotational Semantics (DS) 표현, Borger and Schulte의 Abstract State Machine (ASM) 표현, Watt와 Brown의 Action Semantics (AS) 표현으로 분류할 수 있다.

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[1,8].

ASM(Abstract State Machine) 표현은 자바의 전체 구문으로 확장 가능하지만 많은 구조를 처리하기에는 불완전한 측면이 있다[1,8]. ASM은 자바의 실제 구문과는 다른 추상구문으로 매우 명확하지만 저급 수준이고, 제한된 의미에서만 모듈방식으로 사용된다[1,8].

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[1,8].

본 연구에서는 파이썬에 대한 의미 구조를 표현하기 위해 기존의 작용식을 수정하고 업그레이드해서 파이썬의 정적이고 동적인 의미 구조를 표현하는 작용식 2.0을 제시한다. 다른 객체지향 언어와 차별화된 파이썬의 다양한 기능이 있는데, 동일한 블록으로 지정하기 위해 들여쓰기를 사용하고, 다양한 종류의 컬렉션을 사용하고, 여러 형태의 함수가 있다. 이런 기능들에 대해 정적이고 동적인 의미 구조를 표현하면서, 다른 기존의 연구들보다 구체적으로 의미 구조를 표현하여 언어의 기능을 손쉽게 확장할 수 있고 언어의 융통성을 높여 번역기를 쉽게 구현할 수 있는 정적이고 동적 의미 구조를 제시하고자 한다.

3. 작용식 2.0

파이썬은 블록을 표시하기 위해 들여쓰기(Identation)를 사용한다. 블록을 시작하는 문장들, 예를 들어 if, for, while, def 명령문들의 끝에는 콜론(:)을 사용하고 내부의 블록은 같은 수의 공백문자를 사용한 들여쓰기로 동일한 블록으로 표시한다. 일반적으로 들여쓰기에는 4개의 공백을 사용할 것을

권장하지만, 동일한 수의 공백을 사용하면 동일한 블록으로 인정한다. list, tuple, set, dictionary 같은 다양한 컬렉션을 사용하고 있고, 함수 호출과 관련된 여러 가지 독특한 기능이 있어서 기존의 작용식으로는 의미 구조를 표현하는데 한계가 있어 작용식을 확장하고 수정해 작용식 2.0을 사용해서 의미 구조를 표현한다.

파이썬의 의미 구조를 명세하기 위해 속성 문법을 확장하고 변형하여 정적이고 동적 의미 구조를 표현하는 작용식 2.0을 제시하고, 작용식 2.0에는 Execute equation, Evaluate equation, Eval_rel equation, Eval_par equation, Wait_in equation, 및 Eval_out equation 이 있다.

- **Execute equation:** 파이썬 언어의 각 명령문을 실행하는 동적 명세를 표현하는 절차적인 식
- **Eval_out equation:** 출력문(print 문)에 나오는 변수나 수식의 값을 계산하는 함수적 식
- **Wait_in equation:** 키보드로부터 자료가 입력되기를 기다렸다 자료가 입력되면 입력되는 자료 값을 반환하는 함수적 식
- **Evaluate equation:** 변수의 값을 나타내는 Value(val) 속성(attribute)을 계산하기 위한 함수적 식으로 계산된 속성 값을 반환하는 식
- **Eval_rel equation:** 관계식 연산을 평가해서 그 관계식 값을 반환하는 함수적 식
- **Eval_par equation:** 괄호를 포함하는 수식을 계산해서 그 수식의 값을 반환하는 함수적 식

각 작용식 2.0에서는 네 가지 사건이 발생할 수 있다. **complete**는 작용식 2.0이 오류 없이 실행을 끝내는 경우이고, **diverge**는 작용식 2.0을 수행하면서 오류가 발생하거나 예외 상황이 발생하여 제대로 종료하지 못하는 경우이고, **store**는 작용식 2.0에서 계산한 값을 저장하는 경우이고, **compute**는 작용식 2.0에서 특정한 수식을 계산하는 경우이다.

작용식 2.0에는 8가지 속성들을 사용하고 있다. **out** 속성은 각 작용식 2.0을 실행한 후에 발생하는 결과를 나타내는 속성이고, **val** 속성은 각 비단말(Nonterminal)의 값을 나타내는 속성이고, **name** 속성은 식별자의 이름을 나타내는 속성이고, **type** 속성은 식별자의 형을 명시하는 속성이고, **pri** 속성은 수식에서 연산자의 우선 순위를 지정하기 위한 속성이고, **env** 속성은 비단말의 환경을 나타내는 것으로 함수를 호출하는 환경과 호출되는 환경을 구분하기 위한 속성이고, **env1** 속성은 명령문이 main에 속한 것인지 아니면 클래스에 속한 것인지 구분하기 위한 속성이고, **env2** 속성은 각 환경에 속해 있는 명령문들에 대해 들여쓰기로 블록을 구분 지을 때 사용하는 속성이다.

파이썬은 자바와 같이 완벽한 캡슐화가 아니면서 클래스를 정의하고 객체를 생성하므로 기존의 환경에 덧붙여 환경 설정을 해야 한다. 즉 각 명령문에 대해 환경을 설정할 때 env1, env2로 두 가지의 환경을 설정하는 과정이 필요하다. setEnv1은 각 명령문이 main에 속한 것인지 아니면 클래스에 속한 것인지 구분하기 위해 환경을 설정하는 함수이다. setEnv2(si.env1)는 각 명령문의 env1 환경변수를 넘겨주면서 env2의 환경 변수 값을 지정하는 함수이다.

countWhitespace는 명령문이 시작할 때 명령문 첫 문자 앞에 공백문자 수를 카운트하는 함수로 공백수를 체크해서 val 속성(current_Whitespace.val)에 저장한다.

```
setEnv1(current_Whitespace.val)
    if current_class_name is NULL then
        return(main)
    else return(current_class_name)
    endif
```

3.1 기본 명령문

파이썬의 명령문에 대한 의미구조를 표현하는 작용식 2.0은 다음과 같다.

◀Execute [stmts] → **event** ⟦ complete | diverge ⟧
Execute [⟨s_1 s_2 ,..., s_n⟩ where n ≥1] →
current_Whitespace.val ← countWhitespace(s_1)
s_1.env1 ← setEnv1()
for i=2 to n do
previous_Whitespace.val ← current_Whitespace.val
current_Whitespace.val ← countWhitespace(s_i)
s_i.env1 ← setEnv1()
if current_Whitespace.val is greater than
previous_Whitespace.val then
s_i.env2 ← setEnv2(s_i.env1)
s_i.out ← Execute [⟨s_i⟩]
od

Evaluate equation은 Value 속성(val)을 계산하기 위한 함수적 작용식으로 계산된 속성 값을 반환한 후 그 속성 값을 저장하는(store) 사건이 발생한다[2].

작용식 2.0에서 Evaluate equation을 다음과 같이 수정하여 확장한다. 'exp'가 기존의 기능에 덧붙여 클래스 객체를 생성하는 것인지 함수 호출인지 구별하여 추가로 처리하고, list, tuple, set, Dictionary 같은 컬렉션을 처리하도록 수정한다. collection_make_table은 'exp'가 list, tuple, set, Dictionary 같은 컬렉션일 경우 테이블에 이들 컬렉션에 대해 값을 저장하는 함수이다. lookup 모듈은 변수 이름을 받아서 그 변수에 저장된 값을 반환하는 기능을 수행한다. make_Array는 컬렉션의 값이 하나 이상의 값을 가지므로 그 값들을 배열로 저장하는 함수이다.

◀Evaluate[⟨exp⟩] → **event** 〖 store 〗
Evaluate [⟨exp⟩] →
if ⟨exp⟩ is number then
return(number)
elseif ⟨exp⟩ is variable then
variable.val ← lookup(variable.env, variable.name)
return(variable.val)
elseif ⟨exp⟩ is expression with parenthesis then
exp.val ← eval_par [⟨exp⟩]
return(exp.val)
elseif ⟨exp⟩ is functionCall then
exp.val ← current_func_name,val
return(exp.val)
elseif ⟨exp⟩ is calssConstruct then
exp.val ← class_name,addr
elseif ⟨exp⟩ is collection then
exp.val ← make_Array(exp)
endif

파이썬의 특성에 맞춰 각 명령문에 대해 Execute equation을 다음과 같이 새롭게 수정하여 제시한다.

⟨ 배정문 Execute equation ⟩
Execute [⟨id⟩ = ⟨exp⟩] →
exp.val ← Evaluate [⟨exp⟩]
id.val ← exp.val
if ⟨exp⟩ is collection then
collection_make_table(id.name, id.val)
endif

⟨ if 문의 Execute equation ⟩
Execute [**if** ⟨con⟩ : ⟨s⟩] →
con.val ← Eval_rel [⟨con⟩]
s.env2 ← setEnv2(s.env1)
if con.val then Execute [⟨s⟩] endif
Execute [**if** ⟨con⟩ : ⟨s_1⟩ **else:** ⟨s_2⟩] →
con.val ← Eval_rel [⟨con⟩]
s_1.env2 ← setEnv2(s_1.env1)
if con.val then Execute [⟨s_1⟩] else Execute [⟨s_2⟩]
endif
Execute [**if** ⟨con_1⟩:⟨s_1⟩ **elif** ⟨con_n⟩:⟨s_n⟩ where n≥2]
→con_1.val ← Eval_rel [⟨con_1⟩]
s_1.env2 ← setEnv2(s_1.env1)
if con_1.val then Execute [⟨s_1⟩]
else
for i=2 to n do
con_i.val ← Eval_rel [⟨con_i⟩]
if con_i.val then Execute [⟨s_i⟩]
od
endif
Execute [**if** ⟨con_1⟩:⟨s_1⟩ **elif** ⟨con_n⟩:⟨s_n⟩ where n≥2
else : ⟨s_{nn}⟩] →
con_1.val ← Eval_rel [⟨con_1⟩]
s_{nn}.val ← true
s_1.env2 ← setEnv2(s_1.env1)
if con_1.val then
Execute [⟨s_1⟩]
s_{nn}.val ← false
else
for i=2 to n do
con_i.val ← Eval_rel [⟨con_i⟩]
if con_i.val then
Execute [⟨s_i⟩]
s_{nn}.val ← false
endif
od
endif
if s_{nn}.val then Execute [⟨s_{nn}⟩]

⟨ while 문의 Execute equation ⟩
Execute [**while** ⟨con⟩ : ⟨s⟩] →
s.env2 ← setEnv2(s.env1)
repeat
con.val ← Eval_rel [⟨con⟩]
if con.val then Execute [⟨s⟩]
repeat_start
endif
repeat_end

Choice_element는 리스트에 있는 요소들을 처음부터 하나씩 가져오면서 리스트의 포인터를 다음 요소로 변경하는 함수이다. Exist_element는 리스트의 끝인지 아닌지 판별하는 함수로 리스트의 끝인 경우는 false 아니면 true를 반환한다.

⟨ for 문의 Execute equation ⟩
Execute [**for** ⟨id⟩ **in range** (⟨exp_1⟩): ⟨s⟩] →
id.val ← Evaluate [⟨0⟩]
exp_1.val ← Evaluate [⟨exp_1⟩]
s.env2 ← setEnv2(s.env1)
repeat
if exp_1.val is greater than id.val
then con.val ← true
else con.val ← false
endif

```
    if con.val then Execute [⟨s⟩]
            id.val ← id.val + 1
            repeat_start
    endif
  repeat_end
Execute [for ⟨id⟩ in range (⟨exp1,exp2⟩):⟨s⟩] →
  id.val ← Evaluate [⟨exp1⟩]
  exp2.val ← Evaluate [⟨exp2⟩]
  s.env2 ← setEnv2(s.env1)
  repeat
    if exp1.val is greater than id.val
       then con.val ← true
       else con.val ← false
    endif
    if con.val then Execute [⟨s⟩]
            id.val ← id.val + 1
            repeat_start
    endif
  repeat_end
Execute [for ⟨id⟩ in range (⟨exp1,exp2,exp3⟩):⟨s⟩]→
  id.val ← Evaluate [⟨exp1⟩]
  exp2.val ← Evaluate [⟨exp2⟩]
  exp3.val ← Evaluate [⟨exp3⟩]
  s.env2 ← setEnv2(s.env1)
  repeat
    if exp2.val is greater than id.val
       then con.val ← true
       else con.val ← false
    endif
    if con.val then Execute [⟨s⟩]
            id.val ← id.val + exp3.val
            repeat_start
    endif
  repeat_end
Execute [for ⟨id⟩ in ⟨list⟩ : ⟨s⟩] →
  env2 ← setEnv2(s.env1)
  repeat
   id.val ← Choice_element(list)
   if Exist_element() then
     Execute [⟨s⟩]
     repeat_start
   else
     repeat_end
  endif
```

3.2 입출력문

Wait_in 식은 키보드에서 자료를 입력받는 식으로 자료가 입력되면 입력되는 자료 값을 반환하는 함수적인 식이다.

1) Eval_out equation

Eval_out equation은 출력문(print 문)에 나오는 변수나 수식의 값을 계산하는 함수적 식이고, 문자와 문자열을 처리할 수 있도록 Eval_out equation을 확장하여 수정한다.

```
◀Eval_out[⟨exp⟩] → event ⟦ store ⟧
Eval_out [⟨exp⟩] →
if ⟨exp⟩ is number then
   return(number)
elseif ⟨exp⟩ is variable then
   variable.val ← lookup(variable.env, variable.name)
   return(variable.val)
elseif ⟨exp⟩ is an expression with parenthesis
then exp.val ← Eval_par [⟨exp⟩]
   return(exp.val)
elseif ⟨exp⟩ is an ⟨exp⟩ then
   exp.val ← Evaluate [⟨exp⟩]
   return(exp.val)
elseif ⟨exp⟩ is character then
   return(character)
elseif ⟨exp⟩ is string then
   return(string)
elseif ⟨exp⟩ is collection then
 return(find_value_collection(exp.name,exp.env1,
        exp.env2)
endif

⟨입력문의 Execute equation⟩
Execute [⟨id⟩ = input()] →
  id.val ← wait_in[⟨id⟩]
Execute [⟨id⟩ = input("⟨string⟩")] →
  Execute [print_out("⟨string⟩")]
  id.val ← wait_in[⟨id⟩]
```

파이썬에서는 "**end**"를 사용해 프린트의 마지막 출력 상태를 제어할 수 있다. 보통의 경우 줄바꿈을 실행하지만 end=" "와 같이 하나의 공백만 주면 줄을 바꾸지 않고 같은 줄에 출력하도록 지정할 수 있고, 출력문에서 "*" 다음에 숫자를 넣으면 그 횟수만큼 문자열을 반복해 출력할 수 있다.

파이썬의 다양한 출력문을 실행할 수 있도록 Execute equation을 다음과 같이 확장한다.

find_value_collection은 출력할 변수가 컬렉션일 경우 컬렉션을 찾아주는 함수로 컬렉션의 값을 하나의 값처럼 출력할 수 있다. 파이썬에서 한 줄 주석문을 넣을 경우 '#'으로 시작한다.

```
z=["가", "나", "다"]     # list z generates
print(z*2)
# print ['가', '나', '다', '가', '나', '다']
t = ("꽃", 5, True)
# tuple t generates
print(t*2)
# print ('꽃', 5, True, '꽃', 5, True)
myset = { 2, 1, 5, 7, 5 }
# set myset generates
print(myset)          # print {1, 2, 5, 7}
```

〈출력문의 Execute equation〉

Execute [**print**(〈out〉)] →
  if out is not list then
    out.val ← Eval_out[〈out〉]
    print_out(out.val)
  else print_out(find_value_collection(out))
  endif
  print_out(line_feed)
Execute [**print**(〈out〉, **end=**'〈char〉')] →
  if out is not list then
    out.val ← Eval_out[〈out〉]
    print_out(out.val)
  else print_out(find_value_collection(out))
  endif
  print_out(Eval_out[〈char〉])
Execute [**print**(〈out〉*〈exp〉)] →
  exp.val ← Eval_out[〈exp〉]
  for i=1 to n do
    out.val ← Eval_out[〈out〉]
    if 〈out〉 is list or tuple then
      print_out(find_value_collection(out))
    endif
  od
Execute[**print**(〈out_1〉+〈out_2〉+...+〈out_n〉 where n≥1)] →
  for i=1 to n do
    print_out(Eval_out[〈out_i〉])
  od
Execute[**print**(〈out_1〉,〈out_2〉,...,〈out_n〉 where n≥1)] →
  for i=1 to n do
    print_out(Eval_out[〈out_i〉])
    print_out(" ")
  od

파이썬은 함수와 관련된 기능이 풍부한 언어로 가변의 파라매터를 사용한다든가 초깃값을 설정하여 호출하고 여러 개의 값을 동시에 반환할 수 있는 함수 등의 다양한 함수 호출 기능이 있다. set_param($param_i$.name, func_name.env, $param_i$.val)는 함수의 파라매터에 초깃값을 지정하는 함수이고, makeArray_table(param.name, func_name.env)는 가변의 파라매터들에 대해 배열로 만들어 저장하는 함수로 파라매터 목록 안에 "*"를 사용하면 함수 호출문에서 파라매터를 넘겨줄 때 가변의 개수만큼 파라매터를 넘겨줄 수 있다.

```
def sumNum(*number):
    sum= 0
    for num in number:
        sum += num
    return sum
sum = sumNum(1,5)
print("sum = ",sum)
sum= sumNum(1,3,2,5)
print("sum = ",sum)
```

〈함수의 Execute equation〉

Execute[**def** 〈func_name〉(〈$parm_1$〉,〈$parm_2$〉,...,〈$parm_n$〉) : where n≥1 〈s〉] →
  s.func_env ← func_name
  for i=1 to n do
    make_table($parm_i$.name, func_name.env)
  od
  func_name.addr ← current_point
Execute[**def** 〈func_name〉(〈$parm_1$〉=〈exp_1〉,〈$parm_2$〉=〈exp_2〉,...,〈$parm_n$〉=〈exp_n〉):where n≥1 〈s〉]→
 s.func_env ← func_name
 for i=1 to n do
   make_table($parm_i$.name, func_name.env)
   $parm_i$.val ← Evaluate [〈exp〉$_i$]
   set_param($parm_i$.name, func_name.env, $parm_i$.val)
 od
 func_name.addr ← current_point
Execute [**def** 〈func_name〉(*〈parm〉) : 〈s〉]→
  makeArray_table(parm.name, func_name.env)
  func_name.addr ← current_point
  s.func_env ← func_name
Execute[〈func_name〉(〈$parm_1$〉,〈$parm_2$〉,..., 〈$parm_n$〉) where n ≥1] →
  for i=1 to n do
    $parm_i$.env ← func_name.env
    $parm_i$.val ← Eval_out[〈$parm_i$〉]
  od
  return_save(func_name.addr)
  control_transfer(func_name.addr)

다음 **sumNumC** 함수는 파라매터 목록 안에 * 파라매터 사용해 가변 개수의 파라매터를 사용하고 함수를 호출할 때 넘겨준 모든 실인수들의 합을 구해서, 넘겨준 실인수의 수와 합계를 튜플(Tuple) 형식으로 반환한다. 튜플 (4, 11)을 반환한 후 "count, sum = sumNumC(1,3,2,5)" 문장에서 count에는 4, sum에는 11을 각각 대입한다.

```
def sumNumC(*number):
    count = 0
    sum = 0
    for num in number:
        count += 1
        sum += num
    return count, sum
count, sum = sumNumC(1,3,2,5)
print("count = ", count, "sum = ", sum)
```

Execute [⟨id_1⟩, ⟨id_2⟩, ... ⟨id_n⟩ = ⟨func_name⟩
(⟨$parm_1$⟩, ⟨$parm_2$⟩, ..., ⟨$parm_n$⟩) where n ≥1] →
for i=1 to n do
$parm_i$.env ← func_name.env
$parm_i$.val ← Eval_out[⟨$parm_i$⟩]
od
return_save(func_name.addr)
control_transfer(func_name.addr)
Execute [⟨func_s⟩]
for i=1 to n do
id_i.val ← $func\_name_i$.val
od

Execute[**retrun**]→
control_transfer(current_func_name.addr)

Execute [**retrun** ⟨exp_1⟩,⟨exp_2⟩,...⟨exp_n⟩ where n≥]→
for i=1 to n do
exp_i.val ← Eval_out[⟨exp_i⟩]
$current\_func\_name_i$.val ← exp_i.val
od
control_transfer($current\_func\_name_1$.addr)

다음 예제는 Car 클래스를 생성하고 Car 클래스에서 상속받아 자식 클래스인 MyCar를 만드는 예제인데, 클래스 안에 선언된 drive 메소드의 첫 번째 매개변수로 self를 갖고 self는 호출한 객체를 의미한다.

```
class Car:
    def drive(self):
        self.speed = 60
class MyCar(Car):
    def drive(self):
        self.speed = 100
```

⟨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[**def** ⟨meth_name⟩(**self**,⟨$parm_1$⟩,...,⟨$parm_n$⟩)
: where n≥0 ⟨s⟩] →
s.meth_env ← meth_name
for i=1 to n do
make_table($parm_i$.name, meth_name.env)
od
meth_name.addr ← current_point

4. 의미표현법 비교와 파이썬 프로그램 테스트

기존의 의미 표현법과 작용식 2.0을 비교하기 위해 기존 연구 중 while 구문에 대한 세 연구의 표현법을 나타내면 다음과 같다.

- **DS(Denotational Semantics) 표현법**

exec⟦ while (Expr) Stmt ⟧ env scont sto =
$scont_1$ (env[&break ← scont], sto) **where rec**
$scont_1$ = λ(env_1, sto_1).eval ⟦ Expr ⟧ env_1
econt sto **where** econt = λ(val, typ, sto_2).
if val = true
then exec ⟦ Stmt ⟧ env_1 $scont_1$ sto_2
else scont(env, sto_2)

- **ASM(Abstract State Machine) 표현법**

let stm = (while (exp stm_1) **in**
ʃst(stm) = ʃst(exp)
*n*xt(exp) = stm
*n*xt(stm_1) = ʃst(exp)
if task **is** (while (exp) stm_1) **then**
if val(exp) = true **then**
task := ʃst(stm_1)
else
task := *n*xt(task)

• **AS(Action Semantics) 표현법**

```
Execute [ "while" "(" E:Expression ")" S:Statement ]=
  unfolding
    evaluate E then
      check ( the given value is true ) then
      execute S then unfold
    or
      check ( the given value is false ) then
      complete
  trap an unlabeled-break then complete
```

파이썬의 서브셋인 minpy의 연산적 의미[5]로 while 문을 명세하면 다음과 같다. while 문에 대해 조건식을 검사하는 상태와 조건이 참인 경우 while 블록을 실행하는 상태와 반복하는 상태를 연산적 의미로 정의하면서 추상 기계의 상태 전이를 각각 명세하고 있다[5].

• while 조건식을 검사하는 상태

$$\begin{array}{l}\quad\langle \Theta, \Gamma, S|\texttt{for } x \texttt{ in } a_i : \circ_b, a \quad\rangle \\ \Rightarrow \begin{cases}\langle \Theta, \Gamma, S|\texttt{for } x \texttt{ in } a_i : b\ , a_f([a_i]) \rangle & \text{if } \_\_\texttt{next}\_\_ \in \Phi^{\Theta}_{a_i} \\ \langle \Theta, \Gamma, S\ , \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise}\end{cases} \\ \textbf{where } a_f = \Phi^{\Theta}_{a_i}(\_\_\texttt{next}\_\_)\end{array}$$

• while 블록을 실행하는 상태

$$\begin{array}{l}\quad\langle \Theta\ , \Gamma|\gamma_1, S|\texttt{for } x \texttt{ in } a_i : b\ , a\rangle \\ \Rightarrow \langle \Theta \oplus [\gamma_1 \to m], \Gamma|\gamma_1, S|\texttt{for } x \texttt{ in } a_i : \circ_b, b\rangle \\ \textbf{where } m = \Theta(\gamma_1) \oplus [x \to a]\end{array}$$

• while문을 반복하는 상태

$$\begin{array}{l}\quad\langle \Theta, \Gamma, S|\texttt{for } x \texttt{ in } a_i : b, \texttt{raise } a\rangle \\ \Rightarrow \begin{cases}\langle \Theta, \Gamma, S\ , a_{\texttt{None}} \rangle & \text{if } a : \Theta a_{\texttt{StopIteration}} \\ \langle \Theta, \Gamma, S\ , \texttt{raise } a \rangle & \text{otherwise}\end{cases}\end{array}$$

파이썬 3.3의 연산적 의미를 정의한 연구[6]의 경우에는 minpy의 확장된 기능들을 명세하여 while문에 대한 명세가 나와 있지 않아 함수를 호출하는 과정을 규칙 〈k〉로 정의하면서 다음과 같이 명세한다.

```
rule 〈k〉
invoke(obj(_:Int, 〈oattrs〉..."__closure__" |-〉
Closure "__code__" |-〉 Code"__globals__" |-〉
Globals...〈/oattrs〉), M) ~〉 K:K =〉
executeFrame(N, Code, ref(Frame),makeLocals(...),
Globals, Globals ["__builtins__"],makeCells(...),
M) ~〉 return
〈/k〉
〈nextLoc〉 N =〉 N +Int 1 〈/nextLoc〉
〈control〉...
 〈currentFrame〉 〈frameObject〉 Frame:Int =〉 N
 〈/frameObject〉
 C:Bag
 〈/currentFrame〉
 〈cstack〉. =〉 ListItem(call(Frame, C, K))
...
〈/cstack〉
...
〈/control〉
```

의미 구조를 명세하는 대표적인 세 표기법 DS, ASM, AS와 파이썬에 대해 정의한 기존 의미 구조[5,6]와 본 작용식 2.0을 비교했을 때 본 작용식 2.0의 경우, env1, env2를 사용해 파이썬이 main에서 선언된 것인지 아니면 클래스에서 선언된 변수들인지 구분하고, 클래스나 함수 안에서 다시 내부 블록에서 선언된 변수인지 변수의 스코프를 적절하게 표현할 수 있고 반복문과 함수를 호출하여 실행하는 의미 구조를 쉽게 표현하고 보다 구체적으로 명세하고 있어 판독성이 뛰어남을 확인할 수 있다.

작용식 2.0에서 정의된 의미 구조가 실제로 구현가능한 의미 명세인지 입증하기 위해, 작용식 2.0의 의미 구조를 자바로 구현해 네 가지 유형의 프로그램으로 실행시간을 측정해서 Table 1에 나타내었다.

Table 1에서 M1, M2는 함수 사용해 최댓값과 최솟값과 합과 평균을 구하는 프로그램이고, P1, P2는 함수 사용해 급료를 계산하는 프로그램이고, S1, S2는 합계를 구하는 함수를 호출하는 프로그램이고, I1, I2는 내장 함수를 사용해 각 함수 값을 계산하는 프로그램이다. 각 프로그램 유형에 대해 배정문(Assignment statement)과 조건문(Conditional statement)과 반복문(Loop)과 함수(Function)에 대한 실행시간을 측정하였다.

간단한 프로그램에 대한 Table 1의 실행시간을 근거로 시뮬레이션을 통해 Table 2에서 다양한 길이를 갖는 위의 네 가지 유형의 프로그램의 실행시간 추정치를 나타내었다.

기존의 파이썬 의미 구조 연구[5,6]에는 실행시간을 측정하지 않아서 다른 파이썬 의미 구조와 성능 측면에서 객관적으로 서로 비교할 수 없지만 작용식 2.0에 명세된 의미 구조가 번역기를 구현할 수 있는 동적 의미를 명세하고 있고 실제

Table 1. Execution Time of Python Program

(unit: us)

Program Type	Assign. Stmt.	Condition Stmt.	Loop	Function
M1	0.5	1	2	2.5
M2	0.5	1	2	2.5
P1	0.5	1	2	2.5
P2	0.5	1	2	2.5
S1	0.5	1	2	2.5
S2	0.5	1	2	2.5
I1	0.5	1.5	2	2.5
I2	0.5	1.5	2	2.5

Table 2. Execution Time for No. of Program Lines

(unit: us)

Program Type	No. of Program Lines				
	100	500	1000	5000	10000
P1	598	2982	5973	28738	59732
P2	585	2920	5839	29027	57436
P3	572	2856	5672	28360	56392
P4	643	3206	6371	31269	62538

적이고 구체적인 의미 구조라서 번역기로 구현 가능함을 확인할 수 있다.

5. 작용식 2.0의 의미 구조 비교 평가

형식 의미를 비교한 국내외 기존 연구에서는 판독성(Readability), 모듈성(Modularity), 확장성(Extensibility), 융통성(Flexibility) 영역에서 의미 구조의 성능을 평가하고 있다. 작용식 2.0은 작용식을 확장해 파이썬의 의미 구조를 명세한 것이고, 파이썬에 대한 의미 구조[5,6]를 비교 분석한 관련 연구가 없으므로 기존의 파이썬 의미 구조에 대한 객관적인 평가를 수행할 수 없고, 객체지향언어인 자바에 대한 의미 구조를 분석한 기존 연구들과 비교함으로써 작용식 2.0의 효율성을 확인하려고 한다.

기존 연구[8-10]들을 중심으로 Table 3는 자바 언어에 대한 판독성, 모듈성, 확장성, 융통성 영역에서 세 표기법인 DS, ASM, AS을 비교한 결과이다. 기존의 세 가지 표기법에 대한 판정은 형식 의미에 대해 비교하고 연구한 기존 국외 논문[8-10]에서 분석한 내용을 Table 3에 그대로 표시하였다. 기존의 형식 의미 연구[8-10]에서도 판독성, 모듈성, 확장성, 융통성 측면에서 형식 의미 구조의 성능을 평가하고 있음을 알 수 있다.

본 연구에서도 기존 국내외 연구에서 의미 구조를 비교 분석한 방법을 적용해 판독성, 모듈성, 확장성, 융통성의 네 영역에서 대표적인 세 가지 의미 구조 표현법과 작용식 2.0을 비교하여 본 작용식 2.0 의미 구조 명세의 우월성을 입증하도록 한다.

먼저, 판독성의 경우, **ASM**이나 **AS**와는 달리 작용식 2.0의 경우 **if**나 **for**같은 일반적으로 널리 알려진 제어 구문을 사용하여 동적 의미구조를 정확하고 자세하게 표현한다. 다른 언어와 차별화된 특성을 갖는 파이썬 구문의 동적 의미구조를 쉽게 파악할 수 있어 판독성이 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

모듈성을 고려해보면 모듈성이 높은 **AS**와 비교할 때, 작용식 2.0의 경우 기존 연구에서 자바에 대한 형식 의미를 명세하는 작용식[1,2]을 확장하여 파이썬의 의미 구조를 명세하기 위해, 블록을 처리하는 기능과 컬렉션을 처리하는 의미 구조를 명세하는 기능 등 새로운 모듈을 추가하여 파이썬의 의미 구조를 구체적으로 명세하기 위한 모듈을 제시하고 있고, 번역 과정을 수행할 때 각 기능을 수행하는 모듈을 작성함으로써 모듈화가 가능하여 모듈성이 뛰어남을 알 수 있다.

확장성이 높은 **ASM**이나 **AS**와 비교할 때, 기존 논문에서 볼 수 있듯이 작용식 2.0의 경우 OBLA 언어 같은 간단한 객체 처리를 위한 작용식[4]뿐만 아니라 기능이 복잡한 자바의 객체 처리나 예외 처리 등을 포함하여 언어의 기능을 확장한 연구[1,2]에서 의미 구조를 명세하였고, 파이썬의 경우에도, 다른 두 표현법과는 달리 본 논문에서 새롭게 정의하여 명세한 것처럼 countWhitespace, collection_make_table, make_Array 같은 함수를 추가하여 파이썬의 의미 구조를 표현하는 작용식 2.0을 정의함으로써 언어의 기능이 확장된 것에 대해 동적 의미구조를 정확하게 명세할 수 있다. 작용식 2.0은 작용식을 기반으로 파이썬의 의미구조를 명세하기 위해 기능을 확장한 것이므로 자바같은 객체 지향언어에 대해 형식 의미를 명세할 수 있고 본 논문에서 명세했듯이 파이썬 같은 언어의 의미를 표현하고 구현할 수 있어 확장가능성이 아주 높은 의미 구조 명세법이다.

융통성이 높은 **ASM**과 비교할 때, 간단한 객체 지향언어인 OBLA 언어[4]뿐만 아니라 자바와 같은 객체 지향 언어 구문을 명세할 경우[1]에도 작용식의 문법 구문만 변경하여 그 언어의 동적 의미구조를 쉽고 적절하게 명세할 수 있어 융통성 있는 형식 의미 명세법이라는 사실을 알 수 있다. 또한 파이썬과 같은 언어도 **ASM**과는 달리 다양한 속성들을 사용하여 의미 구조를 보다 명확하게 전달할 수 있으므로 새로운 언어에 대한 의미 구조를 구현하는 과정을 더 효율적으로 진행할 수 있다.

제시된 작용 식 2.0은 연산 의미론(Operational semantics)을 근거로 정확하고 실제적인 의미 구조를 제시하였고 무엇보다 간단하고 쉽게 정적인 의미뿐만 아니라 동적인 의미구조도 적절하게 표현하고 있다. 형식 의미를 정확하게 정의하는 목적이 최적화나 번역기를 구현하는 과정에 사용하는 것인데 작용식을 사용하여 구현된 번역기[3,4]를 통해 손쉽게 번역기를 구현할 수 있듯이 기존의 작용식을 확장한 작용식 2.0의 경우도 빠르고 쉽게 번역기를 구현할 수 있도록 의미 구조를 표현하고 있고 번역기를 구현하는 과정에 그대로 활용할 수 있어 보다 실제적이고, 구체적이며, 정확한 형식 의미 구조로 명세된 사실을 파이썬 프로그램들의 실행 시간 테스트를 통해 확인할 수 있다.

6. 결 론

본 논문에서는 파이썬을 연산 의미론에 근거하여 정적이고 동적인 형식 의미를 명세하는 작용식 2.0을 제시하였다. 작

용식 2.0은 작용식[1,2]을 기반으로 파이썬을 위해 의미 명세 기능을 확장한 것으로 명령형 언어와 객체 지향 언어인 자바를 비롯해 파이썬을 위한 정적이고 동적인 의미 구조를 정확하게 명세할 수 있다.

본 논문에서 제시된 작용식 2.0은 연산 의미론에 근거하여 의미 구조를 정의하여 기존의 의미 표현법과 비교할 때 아주 간단하게 의미 구조를 명세할 수 있고, 번역기를 신속하게 구현할 수 있도록 보다 실제적이고 구체적이고 정확한 동적 의미 구조를 제시하고 있다.

작용식 2.0에서 정의된 의미 구조를 자바로 구현해 네 가지 유형의 프로그램의 실행시간을 측정한 값으로 시뮬레이션을 통해 다양한 길이의 프로그램의 대한 실행시간 추정치를 계산함으로써 작용식 2.0에 명세된 의미 구조로 번역기를 구현할 수 있음을 확인할 수 있다.

의미 구조를 표현하는 대표적인 세 표기법인 DS, ASM, AS와 본 논문에서 제시한 작용식(AE) 2.0을 판독성, 모듈성, 확장성, 융통성 영역에서 비교했을 때 모든 영역에서 작용식 2.0이 높이 평가되었고 작용식 2.0이 기존의 대표적인 표현법보다 우수함을 확인할 수 있다.

References

[1] Jung Lan Han, "Formal Semantics for Processing Exceptions," *KIPS Transactions on Computer and Communication Systems*, Vol.17, No.4, pp.173-180, Apr. 2010.

[2] Jung Lan Han, "Specification of Semantics for Object Oriented Programming Language," *KSII Transactions on Internet and Information Systems*, Vol.8, No.5, pp.35-43, 2007.

[3] Jung Lan Han and Sung Choi, "Building of Integrated Increment Interpretation System Based on Action Equations," *KIPS Transactions on Computer and Communication Systems*, Vol.11, No.3, pp.149-156, Mar. 2004.

[4] Jung Lan Han, "Incremental Interpretation Based on Action Equations," Ph. D Thesis Ewha Womans University, 1999.

[5] Gideon Joachim Smeding, "An executable operational semantics for Python," Master's thesis, Utrecht University, 2009.

[6] Dwight Guth, "A Formal Semantics of Python 3.3," Master's thesis, University of Illinois, 2013.

[7] Suhwan Ji, Hyeonseung Im, "Implementing Structural Operational Semantics in Python," *Jorunal of Korean Institute of Information Scientists and Engineers*, Vol.45, No.11, 2018.

[8] Yingzhou Zhang and Baowen Xu, "A Survey of Semantic Description Frameworks for Programming Languages," *ACM SIGPLAN Notices*, Vol.39, No.3, pp.14-30, Mar. 2004.

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

[10] David A. Watt and Deryck 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