액션 의미표기법을 통한 객체의 이해

도 경 구†

요 약

이 논문은 액션 의미표기법을 사용하여 객체와 그에 관련된 연산들의 의미를 정형적으로 정의한다. 액션 의미표기법은 다른 표기법에 비해서 객체연산의 계산과정을 더 명확히 표현할 수 있을 뿐만 아니라, 구현 방식에 대한 힌트도 얻을 수 있다는 장점이 있다. 이를 보여주기 위해서 Abadi-Cardelli의 시그마 계산표기법에 대한 액션 의미구조를 정의하고, 예제 프로그램을 가지고 그 의미를 전개해 본다.

Understanding Objects : An Action Semantics Approach

Kyung-Goo Doh†

ABSTRACT

This article uses action semantics to formally specify the meaning of objects and their related operations. The action-semantics framework, compared to others, is able to not only express object-oriented computation steps more clearly, but also provide a hint on how to implement them. As a showcase, an action semantics of a variant of Abadi-Cardelli's ς -calculus is defined. Then we use an example program to show how to derive the meaning.

1. 서 론

프로그래밍언어를 모호성과 오해의 여지가 없도록 정의하기 위해서는 정형적인 의미표기법을 사용해야 한다. 그러나 정형적으로 정의된 의미구조(semantics)를 보통 수준의 일반 프로그래머에게 이해시키는 데는 현실적으로 무리가 따라 보인다. 왜냐하면, 오랜 기간 동안 널리 알려져 왔던 표시적 의미표기법(denotational semantics)[10, 11]이나 동작적 의미표기법(operational semantics)[3, 9] 조차도 그 표기법을 제대로 이해하기 위해서는 상당히 높은 수준의 훈련이 필요하기 때문이다. 이러한 현실적인 격차를 메우기 위해 개발된 표기법이 액션 의미표기법(action semantics)이다[4-6, 12]. 액션 의미표기법의 모토는 실질적으로 사용하는 프로그래밍언어의 의미구조를 실용적으로 표현할 수 있도록 하자는 것이다[4-6]. 액션 의미표기법에서는 프로그램의 의미를 액션(action)이라고 하는 메타언어로 나타내는데, 값의 전달, 산술, 바인딩의 생성과 참조, 저장장소 할당과 조작 등 정보처리에 관련된 기본적인 동작이 기본 액션으로 정의되어 있고, 정보의 흐름을 제어하는 액션결합자로 액션들을 더 복잡한 액션으로 결합할 수 있다. 특히 액션은 의미를 연상할 수 있는 영어 이름으로 되어 있어서 액션의 정확한 의미를 모르고도, 그 의미를 직관적으로 추측할 수도 있다. 이미 명령형 언어인 Pascal과 함수형 언어인 Standard ML은 이미 액션 의미표기법으로 완전히 정의되었다[8, 13, 14].

이 논문에서는 액션 의미표기법을 사용하여 객체지향언어의 의미를 표현하는 방법을 제시한다. 객체지향언어의 가장 핵심적인 개념과 연산을 갖추고 있으면서 구문구조가 간단한 Abadi-Cardelli의 시그마 계산표기

※ 본 논문은 한국과학재단의 특성기초연구(과제번호 : 2000-1-30300-010-3) 연구비 지원에 의한 것임.
† 정 회 원 : 한양대학교 전자컴퓨터공학부 교수
논문접수 : 2000년 6월 9일, 심사완료 : 2000년 11월 11일

법(ς-calculus)[1]을 가지고 객체의 의미와 메쏘드/필드의 호출과 경신의 의미를 살펴본다. 표시적이나 동작적으로 표기한 의미정의에 비해서 액션으로 표기한 의미는 간결하지는 않지만, 직관적으로 더 이해하기가 쉽고 실제적으로 계산이 이루어지는 과정을 단계별로 이해할 수 있는 장점이 있다. 뿐만 아니라 표기된 액션은 객체 및 메쏘드/필드에 관련된 연산의 구현 방식도 제시하고 있어서 의미의 파악과 함께 구현 방법에 관한 힌트도 얻을 수 있는 장점도 있다. 순수한 시그마 계산표기법만 가지고 객체지향적으로 함수의 정의와 호출의 표현이 가능하지만, 설명을 최대한 간략하게 하기 위해서 람다 계산표기법(λ-calculus)을 함께 사용한다. 프로그램에 의미함수를 적용하여 그 프로그램의 의미를 전개할 수 있는데, 이 논문에서는 객체지향 자연수의 예를 가지고 그 의미를 전개한다. 액션 의미표기법의 확장성, 모듈성 등 다른 장점은 이 논문의 초점에서 벗어나므로 특별히 강조하지는 않는다.

이 논문은 다음과 같이 구성된다. 다음 절에서는 액션 의미표기법을 간단히 소개하고, 예로서 람다 계산표기법의 의미모듈을 정의한다. 3절에서는 Abadi-Cardelli의 시그마 계산표기법[1]의 의미를 액션 의미표기법을 사용하여 정형적으로 정의한다. 4절에서는 3절에서 정의한시그마 계산표기법의 의미정의를 가지고 객체지향으로 정의된 자연수의 의미를 고찰해본다. 그리고 5절에서 결론을 맺는다.

2. 액션 의미표기법

액션 의미표기법[4-6, 12]은 문맥자유문법(context-free grammar)으로 프로그램의 추상구문구조(abstract syntax)를 정의하고, 의미함수(semantic functions)를 통해서 그 구문구조에 의미를 부여한다. 즉, 의미함수는 추상구문구조를 입력으로 받아서, 그 의미를 나타내는 액션을 결과로 내주는 함수이다. 각 의미함수는 표시적 의미표기법[10, 11]과 마찬가지로 합성적(compositional)으로 정의된다. 합성적이란 프로그램의 의미가 그 프로그램을 구성하는 부분프로그램의 의미에 의해서 결정된다는 뜻이다.

액션 의미표기법에 의한 프로그래밍언어 정의모듈은 추상구문구조 모듈, 의미함수 모듈, 의미엔티티(semantic entities) 모듈로 이루어진다. 이 절에서는 먼저 의미를 표현하는 메타언어인 액션표기법(action notation)에 대해서 간단히 소개하고, 람다 계산법의 의미구조를 예로 들어 액션 의미표기법으로 언어 정의모듈을 구성해 본다. 여기서 정의된 액션 의미표기법 언어모듈은 3절에서 정의하는 시그마 계산표기법 정의모듈에 포함되어 사용된다.

2.1 액션표기법

이 절에서는 프로그램의 의미를 표현하는 메타언어인 액션표기법에 대해서 간단히 소개한다.

액션은 구현의 세부사항과는 독립적으로 프로그램의 계산과정을 나타내는 의미엔티티이다. 실제로 액션표기법에 사용되는 의미엔티티의 종류는 액션, 데이터(data), 산출자(yielder)의 3가지가 있다. 액션의 수행은 정보처리과정을 단계별로 어떻게 반영되는지 직접 나타내주고, 액션 수행의 각 단계는 주어진 정보를 취하거나(access) 변경(change)한다. 산출자는 액션의 내부에 포함되는데 주어진 정보를 취할 수는 있지만 변경할 수는 없다. 산출자를 평가하면 항상 데이터가 된다. 예를 들어 the given data는 주어진 데이터를 그대로 산출하고, the data bound to "x"는 주어진 바인딩에서 토큰 "x"에 바인드 되어있는 데이터를 산출한다. 산출자는 액션에 포함되는데, 예를 들어, give the data bound to "x"는 주어진 바인딩에서 토큰 "x"에 바인드 되어있는 데이터를 내주는 액션이고, bind "x" to the given data는 주어진 데이터에 "x"라는 이름을 부여한 바인딩을 생성하는 액션이다. 액션 자체는 데이터가 아니지만 포장하여 abstraction이라는 데이터로 만들 수 있다. 추후에 enact Y(여기서 Y는 abstraction)라는 액션을 사용하여 포장을 풀어 다시 액션으로 되돌려 놓을 수 있다. 여기서 포장하는 작업은 '캡슐화(encapsulation)한다' 라고 하고, 포장을 푸는 작업은 '활성화(enaction)한다' 라고 한다.

액션은 계산이 어떻게 수행되느냐에 따라서 4 가지 상황으로 구분된다. 즉, 액션은 정상적으로 수행되어 끝나거나(complete : 그 액션을 포함하고 있는 액션은 정상적으로 수행이 계속됨), 예외상황이 발생하여 빠져나오거나(escape : 그 액션을 포함하고 있는 액션은 예외처리장치에 도달할 때까지 계속 빠져나감), 현재 수행중인 액션이 실패하거나(fail : 남아있는 대체 액션이 대신 수행됨), 종료하지 않는다(diverge : 그 액션을 포함하고 있는 액션도 종료하지 않음).

액션 수행을 데이터의 생명의 길이에 따라 나누어보

면 다음과 같이 네 가지 종류가 있다. 첫째로 일시 정보를 내주는 give Y 같은 액션이 있다. 이 액션이 내주는 정보는 바로 사용하지 않으면 사라져 버리므로 일시정보라고 한다. 둘째는 데이터의 이름 바인딩을 생성하는 bind I to Y 같은 액션이다. 이 액션이 생성한 바인딩은 영역정보라고 하는데 전체 액션을 통해서 참조가 가능하지만, 부분액션에 의해서 부분적으로 가려질 수 있다. 셋째는 저장장소에 정보를 저장하는 store Y_1 in Y_2 같은 액션이다. 이 액션에 의해서 저장된 정보는 불변정보라고 하는데, 변경할 수는 있지만 가릴 수는 없고, 지우거나 메모리를 해제할 때까지 그대로 살아있다. 넷째로 분산 에이전트끼리 통신하는 액션이 있는데, 이 액션이 만들어내는 정보는 지울 수 조차도 없어서 영구정보라고 한다.

액션이 동작하는 대상은 여러 종류의 양상(facet)으로 구분된다. 기본 양상(basic facet)은 제어흐름을 주로 나타내며 정보와는 독립적으로 처리된다. 함수양상(functional facet)은 데이터 연산 액션이 포함되며 일시정보를 처리한다. 선언양상(declarative facet)은 바인딩 관련 액션이 포함되며 영역정보를 처리한다. 명령양상(imperative facet)은 저장장소 관련 액션이 포함되며 불변정보를 처리한다. 통신양상(communicative facet)은 에이전트의 분산시스템 관련 액션이 포함되며 영구정보를 처리한다. 이 논문에서는 객체의 의미를 함수적으로 표현하기 때문에 명령양상과 통신양상은 취급하지 않는다.

액션은 기본액션(primitive actions)과 액션결합자(action combinators)로 구성되어 있다. 기본액션들은 액션결합자와 결합되어 정보와 제어의 흐름을 조절한다. 어떤 액션 결합자는 수행순서를 정해주기도 하고, 데이터를 결합하거나 전달해 주기도 한다. 이 논문에서 액션표기법에 대하여 더 이상의 세부적인 설명은 생략한다. 사실 액션표기법 자체가 의미를 유추해 낼 수 있을 만큼 표현력이 있는데다가, 본문에 나오는 액션에 대하여 설명이 필요하다고 판단이 되면 그 자리에서 자세히 설명을 하도록 한다. 전체 액션표기법은 Peter Mosses의 책과 논문[4, 7]에 자세히 정형적으로 정의되어 있으므로, 보다 완벽한 액션표기법의 이해를 위해서는 그 책과 논문을 참조하기 바란다.

2.2 액션 의미정의 모듈 : 람다 계산표기법

이 절에서는 액션 의미표기법으로 정의된 람다 계산표기법의 언어정의 모듈인 LambdaCalculus/Action Semantics를 살펴본다. 언어정의 모듈은 추상구문 모듈과 의미함수 모듈과 의미엔티티 모듈로 구성되는데, 각각 따로 정의한 후 다음과 같이 포함(include)한다.

```
module: LambdaCalculus/Action Semantics.
  includes: LambdaCalculus/Abstract Syntax,
            LambdaCalculus/Semantic Functions,
            LambdaCalculus/Semantic Entities.
endmodule: LambdaCalculus/Action Semantics.
```

포함되는 모듈들은 includes : 키워드 뒤에 그 이름을 나열하여 표시하는데, 포함된 모듈은 포함하는 모듈에 귀속된다.

먼저 LambdaCalculus의 추상구문 모듈부터 살펴보자. 추상구문 문법(grammar)은 확장된 BNF(Backus-Naur Form)로 표현된다.

```
module: LambdaCalculus/Abstract Syntax.
  grammar:
  (*) Expr = [[ "lam" "(" Iden ")" Expr ]]
           | [[ Expr "(" Expr ")" ]]
           | Iden.
  (*) Iden = [[ letter (letter | digit)* ]].
  endgrammar.
endmodule: LambdaCalculus/Abstract Syntax.
```

이 문법은 표현식(expression)을 나타내는 Expr 구문도메인과 이름(identifier)을 나타내는 Iden 구문도메인을 정의한다. 여기서 (*)는 단순히 정의의 판독성을 높이기 위해서 사용하는 표시일 뿐 별다른 의미는 없다. 이 문법에서 정의된 표현식에는 람다정의(lambda abstraction), 람다적용(application), 식별자(identifier)가 있다. 문법에서 쌍 대괄호 [[…]]는 …에 있는 개체들을 트리로 구성함을 나타낸다. 예를 들어, [["lam" "(" Iden ")" Expr]]은 심벌 "lam"과 심벌 "("와 비단말자 Iden과 심벌 ")"와 비단말자 Expr의 부분트리 5개로 이루어진 트리를 나타낸다.

의미함수는 표현식 트리를 입력으로 받아 그 의미를 나타내는 액션을 내준다. 액션 의미표기법은 Scott-Strachey의 표시적 의미표기법[11]과 마찬가지로 합성적이어서 프로그램의 의미가 그 프로그램의 부분의 의미에 의해서 결정된다. LambdaCalculus의 의미함수 모듈은 다음과 같다.

```
module: LambdaCalculus/Semantic Functions.
  needs: LambdaCalculus/Abstract Syntax,
         LambdaCalculus/Semantic Entities.
  introduces: evaluate _.
  variables: I:Iden; E,E1,E2:Expr;

   (*) evaluate _::Expr -> action[giving a value].
 [01:] evaluate [[ "lam" "(" I ")" E ]] =
         give closure abstraction of
         (furthermore bind I to the given abstraction
          hence evaluate E).
 [02:] evaluate [[ E1 "(" E2 ")" ]] =
         evaluate E1 then
         enact application of the given abstraction
            to closure abstraction of evaluate E2.
 [03:] evaluate I = enact the abstraction bound to I.

endmodule : LambdaCalculus/Semantic Functions.
```

의미함수 모듈은 추상구문 모듈과 의미엔티티 모듈을 사용하기 때문에, 그 모듈의 이름들이 needs: 키워드 뒤에 나열되어 있다. 그러나, 사용된 모듈은 단순히 참조만 될 뿐, 사용하는 모듈에 귀속되지는 않는다. 구문트리를 나타내 주는 변수는 variables: 키워드 뒤에 해당 구문 도메인 이름과 함께 선언된다. 이 모듈에서 정의된 의미함수의 이름은 introduces: 키워드 뒤에 명시되어 있다. 의미함수 evaluate는 구문도메인 Expr에 속한 구문트리를 값을 내주는 액션으로 매핑한다.

의미식 [01:]에서 [["lam" "(" I ")" E]]는 함수정의를 나타내는 표현식으로, 평가를 미룬 채 함수를 나타내는 데이터로 만들어 주어야 한다. 그렇게 하기 위해서 함수의 의미에 해당하는 액션을 포장하여(abstraction of…), abstraction으로 만든다. 함수 정의의 형식파라미터인 I는 사용영역이 E에만 국한되어야 하므로, 포장 속에 들어가게 되는 액션은 인수를 I에 바인드한 후 주어진 바인딩에 덮어씌우는 액션(furthermore bind I to the given abstraction)과 그 바인딩의 영역 내에서 (hence) E를 평가하는 액션(evaluate E)이 된다. 여기서 closure는 그 곳에서 주어진 바인딩을 포장 속에 포함시켜, 추후에 포장을 풀어 내부의 액션을 수행할 때 그 포함된 바인딩을 제공하도록 하는 역할을 한다. 이는 정적영역규칙(static scoping)을 사용함을 의미한다. 의미식 [02:]에서 [[E1 "(" E2 ")"]]는 함수적용을 나타내는 표현식으로, 먼저 E1을 평가하는데(evaluate E1) 그 결과 값은 포장된 함수정의 abstraction이어야 한다. 이는 then 액션결합사에 의해서 오른쪽 액션으로 넘겨지는데, 오른쪽 액션은 이를 받아서 인수를 적용하며 활성화한다(enact application of … to …). 이 때 인수 E2는 평가가 연기된 채 포장되어 적용된다. 의미식 [03:]에서 볼 수 있듯이, 식별자 I가 참조될 때 비로소 그 연기된 인수는 활성화되어 평가된다. 이는 이름호출 (call-by-name)파라미터 전달방식이 사용됨을 의미한다.

여기서 정의된 언어는 순수 람다 계산표기법이므로 표현식을 평가한 결과 값은 함수값(abstraction)뿐이다. 사실 순수 람다 계산표기법에서 모든 데이터(함수 포함)는 함수값으로 묘사할 수 있다.

의미함수 모듈에서 의미를 나타내는데 사용된 표기들은 아래와 같이 의미엔티티 모듈에 명시하는데, 이 경우 Peter Mosses의 책에 정의되어 있는 표준 데이터 Data Notation[4, 7]을 그대로 사용하고 추가로 정의된 엔티티가 없기 때문에 includes: Data Notation 이라고만 명시한다.

```
module: ExpCore/Semantic Entities.
  includes: Data Notation.
endmodule: ExpCore/Semantic Entities.
```

이 절에서는 람다 계산표기법의 의미구조를 액션 의미표기법으로 정의하여 보았다. 이 액션 정의모듈은 다음 절에서 다루게 될 객체 계산표기법의 정의모듈에 그대로 포함되어 사용된다.

3. 객체 계산표기법의 의미구조

이 절에서는 Abadi-Cardelli의 시그마 계산표기법[1]의 의미를 액션 의미표기법을 사용하여 정형적으로 정의한다. 이 계산표기법은 객체 계산표기법(object calculus)이라고도 하는데, 계산을 수행하는 유일한 기본 골격은 객체(object)이다. 이 계산표기법은 구문구조가 아주 간단함에도 불구하고 객체지향 프로그램에서 필요로 하는 기본개념인 객체생성, 필드(field) 및 메쏘드(method) 호출 및 갱신, 상속(inheritance)을 모두 표현할 수 있어서 객체를 연구하는 도구로 많이 사용된다. 따라서 이 절에서는 액션 의미표기법으로 객체지향 언어의 의미를 어떻게 정형적으로 표현하고 이해할 수 있는지를 알아본다.

3.1 구문구조

이 논문에서 정의하는 객체 계산표기법에서 객체는

이름을 가진 필드와 메쏘드의 집합이며 대괄호로 둘러싼다. 메쏘드는 객체 자신을 나타내는 셀프(self) 변수와 결과를 만들어내는 본체로 구성된다. 필드는 셀프 변수를 사용하지 않는다는 사실만 제외하고 메쏘드와 같다고 보면 된다. 객체 연산에는 필드/메쏘드 호출과 경신이 있다. 순수한 객체 계산표기법만으로 2.2절에서 본 람다 계산표기법을 인코드할 수 있지만[1], 이 논문에서는 판독성을 높이기 위해서 람다 계산표기법을 객체 계산표기법에 포함시켰다. 람다 계산표기법을 포함한 객체 계산표기법의 구문구조는 아래와 같이 정의한다.

```
module: ObjectCalculus/Abstract Syntax.
 includes: LambdaCalculus/Abstract Syntax
 grammar:
 (*) Expr =
       [[ "[" Methods "]" ]] {객체}
     | [[ Expr "." Label ]] {필드/메쏘드 호출}
     | [[ Expr "." Label ":=" Expr ]] {필드경신}
     | [[ Expr "." Label "<="
        "sigma" "(" Iden ")" Expr ]] {메쏘드경신}
     | Iden.
 (*) Methods =
       [[ Label "="
        "sigma" "(" Iden ")" Expr ]] {메쏘드}
     | [[ Label "=" Expr ]] {필드}
     | [[ Methods "," Methods ]].
 (*) Label = [[ letter+ ]].
 endgrammar.
endmodule : ObjectCalculus/Abstract Syntax.
```

위의 구문 정의에서 includes: LambdaCalculus/Abstract Syntax로 명시함으로써 2.2절의 람다 계산표기법 구문구조가 포함되었음을 알 수 있다. 객체 계산표기법의 구문구조를 좀 더 자세히 살펴보자. 객체는 라벨(label)이 서로 다른 메쏘드와 필드의 집합으로서, 나열된 순서는 객체의 의미에 아무런 영향을 미치지 않아야 한다. 어떤 메쏘드가 포함되어 있는 객체를 그 메쏘드의 주인객체라고 한다. 따라서 메쏘드의 "sigma" 뒤의 식별자는 셀프 파라미터로서 그 메쏘드의 주인객체가 바인드되며, 그 영역은 그 메쏘드의 본체가 된다. 필드는 주인객체를 사용하지 않으므로 주인객체 바인딩이 필요없다. 메쏘드와 필드의 호출은 같은 구문구조를 사용하고, 경신은 다른 구문구조를 사용한다.

3.2 의미구조

객체 계산표기법의 의미구조는 다음과 같이 정의할 수 있다.

```
module: ObjectCalculus/Semantic Functions.
  needs: ObjectCalculus/Abstract Syntax.
         ObjectCalculus/Semantic Entities.
  introduces: evaluate _, build an object with _.
  variables: M,M1,M2:Methods; E,E1,E2:Expr;
             L:Label; I:Iden;

  (*) evaluate _::Expr -> action.
[04:] evaluate [[ "[" M "]" ]] = build an object with M.
[05:] evaluate [[ E "." L ]] =
      evaluate E then
      (give the given object at L
       and
       give the given object) then
      ((check (the given tag#1 is field-tag)
        and then
        give the given abstraction#2)
       or
       (check (the given tag#1 is method-tag)
        and then
        enact application of the given abstraction#2
              to the given object#3)).
[06:] evaluate [[ E1 "." L ":=" E2 ]] =
      (evaluate E1 and evaluate E2) then
      give overlay
       (the map of L to
         (field-tag, the given abstraction#2),
        the given object#1).
[07:] evaluate
       [[ E1 "." L "<=" "sigma" "(" I ")" E2 ]] =
       evaluate E1 then
       give overlay
         (the map of L to
          (method-tag,
           the closure abstraction of
            (furthermore bind I to the given object
             hence evaluate E2)),
           the given object).
[08:] evaluate I = give the object bound to I
                   or
                   enact the abstraction bound to I.

  (*) build an object with _::Methods -> action.
[09:] build an object with
        [[ L "=" "sigma" "(" I ")" E ]] =
      give map of L to
        (method-tag,
         the closure abstraction of
          (furthermore bind I to the given object
           hence evaluate E)).
[10:] build an object with [[ L "=" E ]] =
      evaluate E then
      give map of L to (field-tag,
                        the given abstraction).
[11:] build an object with [[ M1 "," M2 ]] =
      (build an object with M1
       and
       build an object with M2) then
      give the disjoint-union(the given map#1,
                              the given map#2).
endmodule : ObjectCalculus/Semantic Functions
```

의미함수 evaluate는 표현식 Expr을 그 표현식의 의미를 나타내는 액션으로 매핑한다. 의미식 [04:]는 객체 표현식의 의미를 나타내는데, 그 객체 표현식이 가지고 있는 메쏘드와 필드에 의해서 그 의미가 결정된다. 메쏘드와 필드의 의미는 의미함수 build an object with에 의해서 표현되는데, 그 의미함수는 메쏘드 또는 필드 구문을 입력으로 받아서 그 의미를 나타내는 액션을 내준다(의미식 [09:], [10:], [11:] 참조). 각 객체의 메쏘드와 필드는 레이블(label)이 있으므로, 레이블을 본체의 의미로 매핑하는 map 데이터로 표현한다. map은 Action Notation[4]에 정의되어 있으며, map of Label to Data 식으로 표현한다. 여기서 Data에는 항상 메쏘드인지 필드인지를 나타내주는 tag이 붙어있어야 한다. tag은 메쏘드/필드 호출을 할 때 메쏘드 본체를 이루는 시그마 정의와 필드 본체를 이루는 람다 정의를 구별하기 위해서 필요하다. 의미식 [09:]는 메쏘드의 의미를 정의한다. 시그마 정의는 평가하지 말아야 하므로 abstraction으로 포장한다. 시그마정의는 객체를 나타내는 셀프 파라미터를 가지고 있으므로, 추후에 메쏘드 호출이나 경신이 이루어질 때 주인객체가 셀프 파라미터에 바인드되게 되고, 이 바인딩의 사용영역은 시그마 정의의 본체가 된다. 의미식 [10:]은 필드의 의미를 정의한다. 필드의 본체는 평가된 후, 그 결과 abstraction이 field-tag과 함께 레이블에 매핑된다. [11:]에서 보여주듯이, 이와 같이 생성된 map들은 disjoint-union 연산으로 결합된다.

의미식 [05:]는 메쏘드 호출의 의미를 나타낸다. 메쏘드 호출 [[E "." L]]은 E를 평가한 결과인 객체를 내준다(evaluate E). 그리고 그 객체에서 L에 해당하는 필드/메쏘드를 골라낸 (give the given object at L) 다음, tag을 조사하여 필드인지 메쏘드인지 구별한다(check). 필드인 경우에는 주어진 람다정의(abstraction)를 활성화하지 않고 그대로 내주고, 메쏘드인 경우 주어진 주인객체를 적용하여 주어진 시그마정의(abstraction)를 활성화한다. 그러면 시그마 정의의 셀프 파라미터에 주인객체가 바인드되고, 메쏘드의 본체가 비로소 평가된다.

의미식 [06:]은 필드 경신의 의미를 표현한다. E1을 평가하여 객체를 내주고, E2를 평가하여 람다정의를 나타내는 abstraction을 내준다. 주어진 람다정의는 L에 매핑되며 주어진 객체에 덮어씌워진다(overlay). 여기서 덮어씌우기는 함수적으로 이루어짐을 주목해야 한다. 다시 말하면, 이 전 저장장소에 있는 내용을 지우고 그 장소에 그대로 내용을 저장하는 게 아니라, 새로운 저장장소를 할당하여 저장한다는 뜻이다. 따라서 경신 이전의 내용은 그대로 남아 있기는 하지만 다시는 참조할 수 없게된다. 이 덮어씌우기는 묵시적으로 이루어진다.

의미식 [07:]은 메쏘드 경신의 의미를 정의한다. E1을 평가하여 객체를 내주고, 시그마 정의를 포장한 abstraction이 method-tag과 함께 L에 매핑된다. 그리고 나서 그 map을 주어진 객체에 덮어씌운다. 여기에서도 마찬가지로 경신이 함수적으로 이루어짐을 주목하라.

의미식 [08:]은 식별자 참조의 의미를 나타낸다. 이 언어에서 식별자는 두 가지가 있는데, 하나는 람다 정의에서 만들어 내는 식별자이고, 다른 하나는 시그마 정의에서 만들어 내는 식별자이다. 이는 각각 불려지면 다르게 쓰인다. 람다 정의 식별자는 캡슐화된 람다 정의가 바인드 되어 있으므로, 호출되면 활성화하여야 하고, 시그마 정의 식별자는 객체가 바인드되어 있으므로 그 객체를 내주기만 하면 된다.

의미를 표현하는 의미엔티티는 다음과 같이 정의한다.

```
module : ObjectCalculus/Semantic Entities.
  includes : Data Notation.
  introduces : object, tag, bindable, label.
  (*) object = map[label to (tag,bindable)].
  (*) tag = field-tag | method-tag.
  (*) bindable = abstraction | object.
  (*) label = string.
endmodule : ObjectCalculus/Semantic Entities.
```

이 절에서는 함수형 언어의 근간이 되는 람다 계산 표기법에 객체지향 개념을 덧붙인 형태의 객체 계산표기법의 의미를 액션표기법으로 정의해보았다. 람다 계산표기법의 의미정의를 그대로 포함하여 사용하여 액션의미표기법의 확장성이 좋음을 확인할 수 있었다.

4. 예제 프로그램과 그 의미 고찰

이 절에서는 3절에서 정의한 객체 계산표기법의 의미정의를 가지고 객체지향으로 정의된 자연수의 의미를 고찰해본다.

4.1 객체지향 자연수

객체지향 자연수는 case 필드와 succ 메쏘드를 가진

객체이다. 0을 나타내는 객체를 정의하면 다른 모든 수는 그 객체를 가지고 정의할 수 있다. 0을 나타내는 객체는 다음과 같이 정의하고, 이를 zero라고 하자.

```
zero = [case = lam(z)lam(s)z,
        succ = sigma(x)x.case := lam(z)lam(s)s(x)]
```

여기서 case 필드는 인수가 2개인 함수 값을 가지는데, 그 함수는 자연수 n에 대해서 n이 0이면 첫 번째 인수를 내주고, 그렇지 않으면 두 번째 인수를 n의 선행수(predecessor)에 적용한다. 따라서, 자연수 1과 2는 각각 다음과 같이 생성되는데, 이를 각각 one과 two라고 한다.

```
one = zero.succ
    = zero.case := lam(z)lam(s)s(zero)
    = [case = lam(z)lam(s)s(zero),
       succ = sigma(x)x.case := lam(z)lam(s)s(x)]
two = one.succ
    = one.case := lam(z)lam(s)s(one)
    = [case = lam(z)lam(s)s(one),
       succ = sigma(x)x.case := lam(z)lam(s)s(x)]
```

여기서 동등성(=)은 객체 계산표기법의 축약(reduction) 규칙에 의해서 보장된다.

이런 식으로 정의된 각 자연수의 선행수를 구하는 프로그램은 다음과 같이 작성할 수 있다.

```
pred = lam(n)n.case(zero)(lam(p)p)
```

즉, two의 선행수는 다음과 같이 구한다.

```
pred(two) = two.case(zero)(lam(p)p)
          = (lam(z)lam(s)s(one))(zero)(lam(p)p)
          = (lam(p)p)(one)
          = one
```

여기서도 동등성은 객체 계산표기법과 람다 계산표기법의 축약규칙에 의해서 보장된다.

4.2 객체지향 자연수 zero의 의미 전개

이제 객체 계산표기법으로 표기된 자연수의 의미를 액션의미표기법으로 전개해보자. 의미 전개 과정에서 =>의 우측에 적용하는 의미식 번호를 표시하고, 의미식 번호가 없는 경우는 액션 수행의 한 단계를 나타낸다.

```
evaluate "zero"
= evaluate
   [[ [case = lam(z)lam(s)z,
       succ = sigma(x)x.case := lam(z)lam(s)s(x)] ]]
=> [04:]
build an object with
   [[ case = lam(z)lam(s)z,
      succ = sigma(x)x.case := lam(z)lam(s)s(x) ]]
=> [11:]
(build an object with [[ case = lam(z)lam(s)z ]]
 and
 build an object with
   [[ succ = sigma(x)x.case := lam(z)lam(s)s(x) ]])
then give the disjoint-union of
          (the given map#1,the given map#2).
```

여기서 case 필드의 의미를 알아보자.

```
build an object with [[ case = lam(z)lam(s)z ]]
=> [10:]
evaluate [[ lam(z)lam(s)z ]] then
give the map of "case" to
            (field-tag, the given abstraction)
=> [01:]
give closure abstraction of
 (futhermore bind "z" to the given abstraction
  hence
  give closure abstraction of
  (furthermore bind "s" to the given abstraction
   hence enact the abstraction bound to "z"))
then give map of "case" to
              (field-tag, the given abstraction)
 =>
give map of "case" to
 (field-tag,
  closure abstraction of
   (futhermore bind "z" to the given abstraction
    hence
    give closure abstraction of
     (furthermore bind "s" to the given abstraction
      hence enact the abstraction bound to "z")))
 = give map of "case" to (field-tag,Y1)
```

이 액션은 case 필드의 의미를 나타낸다.

```
build an object with
   [[ succ = sigma(x)x.case := lam(z)lam(s)s(x) ]]
=> [09:]
give map of "succ" to
 (method-tag,
  closure abstraction of
   (furthermore bind "x" to the given object
    hence
    evaluate [[ x.case:= lam(z)lam(s)s(x) ]])
    => [06:]
```

```
    (evaluate "x"
     and
     evaluate [[ lam(z)lam(s)s(x) ]])
    then
    give overlay
     (the map of "case" to
           (field-tag, the given abstraction#2),
      the given object#1)
여기서
  evaluate "x"
  => [08:]
  give the object bound to "x"
그리고
  evaluate [[ lam(z)lam(s)s(x) ]]
  => [01:]
  give closure abstraction of
  (furthermore bind "z" to the given abstraction
   hence
   give closure abstraction of
    (furthermore bind "s" to the given abstraction
     hence
     (enact the abstraction bound to "s" then
      enact application of the given abstraction
        to closure abstraction of
            give the object bound to "x")))
```

이 두 결과를 종합하여 위의 식을 계속 전개해 보면,

```
=>
give overlay
  (the map of "case" to
  (field-tag,
   closure abstraction of
   (furthermore bind "z" to the given abstraction
    hence
    give closure abstraction of
      (furthermore bind "s" to the given abstraction
       hence
       (enact the abstraction bound to "s" then
        enact application of the given abstraction
          to closure abstraction of
            give the object bound to "x")))),
   object bound to "x")
=>
give map of "succ" to
 (method-tag,
  closure abstraction of
  (furthermore bind "x" to the given object
   hence give overlay
   (the map of "case" to
    closure abstraction of
    (furthermore bind "z" to the given abstraction
     hence
     give closure abstraction of
     (furthermore bind "s" to the given abstraction
      hence
      (enact the abstraction bound to "s" then
       enact application of the given abstraction
         to closure abstraction of
           give the object bound to "x"))),
     the object bound to "x"))
= give map of "succ" to (method-tag,Y2)
```

이 액션은 succ 메쏘드의 의미를 나타낸다.

이 두 결과를 가지고 위의 식을 계속 전개해 보면,

```
=>
give the disjoint-union
          (map of "case" to (field-tag,Y1),
           map of "succ" to (method-tag,Y2))
= give ZERO
```

위의 최종 액션이 생성된 zero 객체의 의미를 나타낸다.

4.3 객체지향 자연수 one의 의미 전개

자연수 1은 zero.succ으로 만들어내는데, 그 의미를 액션으로 전개해 보면 다음과 같다.

```
evaluate [[ zero.succ ]]
=> [05 : ]
evaluate "zero" then
(give the given object at "succ"
 and give the given object) then
((check (the given tag#1 is field-tag) and then
  give the given abstraction#2)
 or
 (check (the given tag#1 is method-tag) and then
  enact application of the given abstraction#2
                to the given object#3))
=>
(give (method-tag,Y2) and give ZERO) then
 ((check (the given tag#1 is field-tag) and then
   give the given abstraction#2)
  or
  (check (the given tag#1 is method-tag) and then
   enact application of the given abstraction#2
                to the given object#3))
=>
enact application of Y2 to ZERO
=>
(furthermore bind "x" to ZERO
 hence
 give overlay
  (the map of "case" to
   (field-tag,
    closure abstraction of
     (furthermore bind "z" to the given abstraction
```

```
      hence
      give closure abstraction of
       (furthermore
          bind "s" to the given abstraction
        hence
        (enact the abstraction bound to "s" then
         enact application of the given abstraction
               to closure abstraction of
                 give the object bound to "x"))),
   the object bound to "x"))
=>
give overlay
 (the map of "case" to
  (field-tag,
   closure abstraction of
    (furthermore bind "z" to the given abstraction
     hence
     give closure abstraction of
      (furthermore bind "s" to the given abstraction
       hence
       (enact the abstraction bound to "s" then
        enact application of the given abstraction
                        to closure abstraction of
                          give ZERO))),
 ZERO))
= give overlay
     (the map of "case" to (field-tag,Y3),ZERO)
= give disjoint-union
     (the map of "case" to (field-tag,Y3),
      the map of "succ" to (method-tag,Y2))
```

위의 최종 액션은 자연수 1을 나타내는 객체인데, 0을 나타내는 객체와의 차이점은 case 필드가 바뀌었다는 것이다. 즉, 1을 나타내는 객체의 succ 메쏘드는 0을 나타내는 객체에서 상속받았음을 알 수 있다.

이 절에서는 간단한 예제 프로그램을 통하여 필드와 메쏘드의 호출과 경신이 어떻게 이루어지는 지 살펴보았다. 언뜻 보아서 축약규칙을 사용하여 유도하는 것보다 훨씬 복잡해 보이기는 하지만, 축약규칙에서 사용되는 구문적 변환과는 달리, 필드와 메쏘드의 호출과 경신이 이루어지는 절차가 의미적으로 정확하게 표현되고 있음을 알 수 있다.

5. 결 론

이 논문에서는 객체와 그와 관련된 연산들의 의미를 액션 의미표기법을 통해서 표현해 보았다. 액션으로 의미를 표기함으로서, 객체와 관련된 연산이 이루어지는 절차를 의미적으로 파악할 수 있을 뿐 아니라, 구현 방식에 관한 힌트도 얻을 수 있었다.

이 논문에서 메쏘드와 필드의 경신은 모두 함수적으로 이루어졌다. 즉, 본문에서 다룬 객체지향언어는 함수형이다. Java[2]와 같은 객체지향언어는 경신이 함수적이 아니라 명령형(imperative)으로 이루어지기 때문에, 명령형 객체지향 언어의 의미를 액션 의미표기법으로 표기해서 그 의미를 고찰해 볼 필요도 있다. 명령형으로 만들기 위해서는 메쏘드나 필드가 경신될 때 객체의 클론(clone)을 만들어야 하므로, 그의 의미를 액션표기법으로 정의할 수 있어야 한다. 이는 추후 연구과제로 남겨둔다.

참 고 문 헌

[1] Martin Abadi and Luca Cardelli, *A Theory of Objects*, Springer, 1996.

[2] K. Arnold and J. Gosling, *The Java™ Programming Language*, Addison-Wesley, 1996.

[3] G. Kahn, Natural semantics, In *STACS'87, Lecture Notes In Computer Science 247*, Springer, Berlin, pp.22-39, 1987.

[4] Peter D. Mosses, *Action Semantics*, No. 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[5] Peter Mosses, A tutorial on action semantics, Tutorial notes for *FME'94* (*Formal Methods Europe, Narcelona, 1994*) and *FME'96* (*Formal Methods Europe, Oxford, 1996*), March 1996.

[6] Peter D. Mosses, Theory and practice of action semantics, In *MFCS'96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland, Lecture Notes in Computer Science*, Vol.1113, pp.37-61, Springer-Verlag, 1996.

[7] Peter D. Mosses, A modular SOS for action notation, Technical Report BRICS RS-99-56, University of Aarhus, December 1999.

[8] Peter D. Mosses and David A. Watt, Pascal Action Semantics, Draft, Version 0.6, May 1993.

[9] Gordon Plotkin, Structural operational semantics, Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[10] David Schmidt, *Denotational Semantics : A Methodology for Language Development*, Allyn and

Bacon, 1986.

[11] Joseph E. Stoy, *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[12] David Watt, *Programming Language Syntax and Semantics*, Prentice-Hall, 1991.

[13] David Watt, The static and dynamic semantics of Standard ML, In *IWAS'99, 2nd International Workshop on Action Semantics, BRICS NS-99-3*, pp.155-172, University of Aarhus, 1999.

[14] David A. Watt, Standard ML Action Semantics, Draft, Version 0.5, May 1997.

도 경 구

e-mail : doh@cse.hanyang.ac.kr

1980년 한양대학교 산업공학과 (학사)

1987년 미국 Iowa State University 전산학과(석사)

1992년 미국 Kansas State University 전산학과 (박사)

1993년~1995년 일본 University of Aizu 교수

1995년~현재 한양대학교 전자계산학과 교수

관심분야 : 프로그래밍언어, 정형기법를 통한 프로그램 검증, 스마트카드