액션의미방식에 의한 언어모듈의 정의와 확장

(Defining and Extending Language Modules: An Action Semantics Approach)

도 경 구 †

(Kyung-Goo Doh)

요 약 언어의 의미정의모듈은 서로 밀접하게 관련 있는 개념과 연산의 의미구조를 모아 놓은 집합이다. 이 논문은 액션 의미표기법으로 의미정의모듈을 구성하고 확장하는 방법을 제시한다. 표현중심언어 핵심 모듈을 먼저 정의하고, 바인딩, 블록구조, 파라미터, 고차 표현식(함수)에 대한 확장 모듈을 정의한다. 그리고 의미의 획일성과 직교성이 보장되도록 의미정의 모듈들을 합성하면 더 복잡한 언어를 구축할 수 있음을 보인다.

Abstract A language module is the collection of language constructs whose concepts and operations are closely related. This paper demonstrates how to use action semantics to define and extend language modules. We first define a language module for an expression language core, and then language modules for bindings, block structures, parameters, and higher-order expressions. Finally, we show that the language modules can be combined, if there is no violation of uniformity and orthogonality, to become a more complex language module.

1. 서 론

표시적 의미표기법(Denotational semantics)[1,2]은 프로그래밍언어의 다양한 양상을 연구하는 수단으로 많은 학자들로부터 각광을 받아왔다. 표시적 의미표기법에서 사용하는 의미 표현을 위해 사용하는 메타언어인 람다 표기법(λ-notation)은 이미 꽤 진보된 도메인 이론의 강한 지지를 바탕으로 프로그램의 수학적인 의미를 정연하게 표현할 수 있을 뿐 아니라, 프로그램의 성질을 증명하는 도구로 적합하다고 널리 알려져 있다. 그러나, 표시적 의미표기법을 실용적으로 사용하기에는 취약한 점이 많이 있다. 프로그램의 의미를 표현하는 고차함수는 바인딩의 영역, 저장장소 등 프로그래밍 언어의 기본 개념과는 거리가 멀다. 게다가, 언어의 설계에 사용하는 경우, 설계 변경이나 확장에 유연하게 대응하지 못하고, 기존에 이미 정의된 부분을 완전히 재작성 해야 하는 단점이 있다. 예를 들어, 직접형(direct-style)으로 작성된 정의에다 도약(jump) 구조를 추가하는 경우 정의 전체를 계속형(continuation-style)으로 재작성 해야 하고, 비결정(nondeterminism)구조를 추가하는 경우 일반 도메인을 파우어 도메인으로 재작성 해야 한다. 더구나, 이미 정의된 부분을 관련된 언어를 정의하는데 재사용하기도 어렵다. 따라서, 표시적 의미표기법을 프로그래밍언어 설계 도구로 실용적으로 사용하기는 적합하지 않다.

이러한 문제점들을 해결하기 위하여 Peter Mosses와 David Watt가 개발한 방법이 액션 의미표기법(action semantics)이다 [3,4,5,6,7]. 액션 의미표기법의 모토는 실질적으로 사용하는 프로그래밍언어의 의미구조를 실용적으로 표기할 수 있도록 하자는 것이다 [3]. 액션 의미표기법에서는 프로그래밍언어의 의미를 나타내기 위하여 액션표기법(action notation)이라는 고수준의 표기법을 사용한다. 값의 전달, 산술, 바인딩의 생성과 참조, 저장장소의 할당과 조작 등 정보 처리와 관련된 기본적인 동작이 기본 액션으로 정의되어 있다. 액션은 정보의 흐름을 제어하는 결합자(combinator)로 결합할 수 있다. 그리고 각 액션은 의미를 연상할 수 있는 영어 이름으로 되어 있어서 초보자라 하여도 직관적으

· 본 연구는 정보통신우수대학지원사업의 일환으로 수행되었음.

† 정 회 원 : 한양대학교 전자컴퓨터공학부 교수

doh@cse.hanyang.ac.kr

논문접수 : 2000년 5월 31일

심사완료 : 2000년 7월 4일

로 그 의미를 추측할 수 있다. 더구나, 액션의미정의는 모듈로 나누기가 용이하다. 따라서, 언어 설계 변경을 적용하여 언어의미정의를 확장하거나 변경하기가 쉽고, 이미 가지고 있는 언어의 정의의 일부를 유사한 관련 언어를 정의하는데 재사용할 수도 있다. Pascal[8]과 Standard ML[9,10] 같은 대형 범용언어가 이미 액션의미표기법으로 완전히 정의되었다.

이 논문에서는 액션 의미표기법에 기반한 언어 정의를 모듈화 하여, 새로운 언어를 설계하는 과정에서 체계적으로 언어 정의를 확장하고 합성하는 방법을 제시한다. 언어의 의미정의모듈은 서로 밀접하게 관련 있는 개념과 연산의 의미구조를 모아 놓은 집합이다. 표현중심언어 핵심 모듈을 기반으로 하여, Landin-Tennent의 프로그래밍언어 확장 원칙[11]을 적용하여 바인딩, 블록구조, 파라미터, 고차 표현식(함수)에 대한 확장모듈을 정의한다. 추상화 원칙(abstraction principle)을 적용하여 표현식 바인딩 모듈을 정의하고, 제한 원칙(qualification principle)을 적용하여 표현식 블록 모듈을 정의하고, 파라미터화 원칙(parameterization principle)을 적용하여 표현식 파라미터 모듈을 정의한다. 각 모듈을 정의하면서, 적극적 바인딩(eager binding)과 소극적 바인딩(lazy binding), 정적 영역결정(static scoping)과 동적 영역결정(dynamic scoping) 방식, 값호출(call-by-value)과 이름호출(call-by-name) 파라미터 전달 방법의 의미구조를 비교 고찰한다. 그리고, 정의한 확장모듈 중에서 서로 밀접히 관련된 모듈들을 의미의 획일성이 보장되도록 잘 조합하면, 함수형 언어모듈을 정의할 수 있음을 보여준다. 액션의미구조는 모듈성과 확장성이 좋기 때문에 이미 존재하는 모듈의 확장에 사용하는 경우 거의 수정이 필요 없다.

이 논문은 다음과 같이 구성된다. 다음 절에서는 액션 의미표기법을 간단히 소개하고, 표현중심언어의 핵심 모듈을 정의한다. 3-5절에서는 추상화 원칙, 제한 원칙, 파라미터화 원칙을 적용하여 언어모듈을 정의한다. 6절에서는 앞 절의 모듈을 조합하여 일차 함수형 언어를 정의한다. 7절에서는 함수의 표현이 가능한 람다계산표기법의 모듈을 설계하고, 이를 포함시켜 고차 함수형 언어를 정의한다. 그리고 마지막으로 8장에서 결론을 맺는다.

2. 액션 의미표기법

액션 의미표기법은 문맥자유문법으로 프로그램의 추상구문트리를 정의하고, 의미함수로 그 트리에 의미를 부여한다. 각 의미함수는 합성적(compositional)으로 정의되며, 추상구문트리가 그 트리의 의미를 나타내는 액션으로 매핑된다. 이 절에서는 액션표기법(action notation)을 간단히 살펴보고, 표현중심언어의 핵심 모듈을 정의한다. 이 모듈은 이 논문에서 제시하는 확장모듈들의 기초가 된다.

2.1 액션표기법

액션은 구현의 세부사항과는 독립적으로 프로그램의 계산과정을 나타내는 의미엔티티(semantic entities)이다. 액션은 정상적으로 수행되어 끝나거나(그 액션을 포함하고 있는 액션은 정상적으로 수행이 계속됨), 예외상황이 발생하여 빠져 나오거나(그 액션을 포함하고 있는 액션은 예외처리장치에 도달할 때까지 계속 빠져나감), 현재 수행중인 액션이 실패하거나(남아있는 대체 액션이 대신 수행됨), 종료하지 않는다(그 액션을 포함하고 있는 액션도 종료하지 않음). 액션 수행을 크게 나누면 다음과 같이 네 가지 종류가 있다. 임시 데이타를 처리하거나, 데이타의 이름 바인딩을 구축하고 참조하거나, 저장장소를 할당하고 조작하거나, 분산 에이전트끼리 통신한다. 액션이 동작하는 대상은 여러 종류의 양상(facet)으로 구분된다. 기본 양상(basic facet)은 제어흐름을 주로 나타내며 정보와는 독립적으로 처리된다. 함수양상(functional facet)은 데이타 연산 액션이 포함되며 일시정보(transient information)를 처리한다. 선언양상(declarative facet)은 바인딩 관련 액션이 포함되며 영역정보(scoped information)를 처리한다. 명령양상(imperative facet)은 저장장소 관련 액션이 포함되며 불변정보(stable information)를 처리한다. 통신양상(communicative facet)은 에이전트의 분산시스템 관련 액션이 포함되며 영구정보(permanent information)를 처리한다. 이 논문은 언어모듈을 구성하고 확장하는 방법에 초점을 맞추고 있기 때문에 명령 양상과 통신양상 액션은 취급하지 않는다. 정의하는 언어의 기초 개념을 이해하는데 중요하다고 여겨지는 액션은 의미모듈을 정의하면서 설명하도록 한다. 그러나, 이름만 가지고 그 의미를 쉽게 파악할 수 있는 액션은 추가적인 설명을 하지 않는다. 전체 액션표기법은 Peter Mosses의 책과 논문에 자세히 정형적으로 정의되어 있다 [3,7].

이 논문에서는 동적의미(dynamic semantics)만 고려하고 있지만, 언어의 정의에는 타입규칙과 같은 정적의미(*static semantics*)의 정의도 상당히 중요하다. 정적의미는 논리학의 추론규칙 형태로 표기하기도 하고 [11], 액션표기법으로 표기하기도 하고 [4,8], 언어가 정적으로 타입을 결정할 수 있는 경우에는 자동으로 추출할 수도 있다 [12,13].

2.2 액션의미정의 모듈: 표현중심언어 핵심

이 절에서는 액션 의미표기법을 사용하여 표현중심언어 핵심(expression language core)의 언어모듈 ExpCore/Action Semantics를 정의한다. 이 언어모듈은 이 논문에서 다루는 다양한 언어모듈의 기초가 된다. 언어모듈(language module)은 추상구문(abstract syntax) 모듈과 의미함수(semantic functions) 모듈과 의미엔티티(semantic entities) 모듈로 구성되는데, 다음과 같이 따로 정의한 후 포함(include)한다.

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

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

먼저 ExpCore의 추상구문 모듈부터 살펴보자.

```
module: ExpCore/Abstract Syntax.
  grammar:
  (*) Expr = Num
           | [[ Expr "+" Expr ]]
           | [[ Expr "=" Expr ]]
           | [[ "not" Expr ]]
           | [[ "if" Expr "then" Expr "else" Expr ]]
           | [[ "(" Expr ")" ]].
  (*) Num = [[ digit+ ]].
  endgrammar.
endmodule: ExpCore/Abstract Syntax.
```

이 문법은 표현식(expression)을 나타내는 Expr 구문도메인과 수(numeral)를 나타내는 Num 구문도메인을 정의한다. 여기서 정의된 표현식에는 수, 덧셈 연산 표현식, 논리 연산 표현식, 조건표현식이 있다. 문법에서 [[...]]는 ...에 있는 개체들의 트리를 나타낸다. 예를 들어, [[Expr "+" Expr]]은 비단말자 Expr과 심벌 "+"와 비단말자 Expr의 부분트리 3개로 이루어진 트리를 나타낸다.

의미함수는 프로그램 트리를 그 의미를 나타내는 액션으로 매핑한다. 액션 의미표기법은 Scott-Strachey의 표시적 의미표기법과 마찬가지로 합성적(compositional)이어서 프로그램의 의미가 그 프로그램의 부분의 의미에 의해서 결정된다. ExpCore의 의미함수 모듈은 다음과 같다.

```
module: ExpCore/Semantic Functions.
  needs: ExpCore/Abstract Syntax,
         ExpCore/Semantic Entities.
  introduces: evaluate _.
  variables: N:Num; E,E1,E2,E3:Expr;
(*) evaluate _ :: Expr -> action[giving a value].
[1:] evaluate N = give decimal string-of-characters N.
[2:] evaluate [[ E1 "+" E2 ]] =
       (evaluate E1 and evaluate E2)
       then give the sum of (the given number#1,
                             the given number#2).
[3:] evaluate [[ E1 "=" E2 ]] =
       (evaluate E1 and evaluate E2)
       then give (the given number#1 is
                  the given number#2).
[4:] evaluate [[ "not" E ]] =
       evaluate E then
       give not the given truth-value.
[5:] evaluate [[ "if" E1 "then" E2 "else" E3 ]] =
       evaluate E1 then
       ( (check the given truth-value
          and then evaluate E2)
         or
          (check not the given truth-value
           and then evaluate E3) ).
[6:] evaluate [[ "(" E ")" ]] = evaluate E.
endmodule: ExpCore/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 프로그램트리를 값을 내주는 액숀으로 매핑한다. 표현식이 내주는 값은 수(number)와 참값(truth-value)이며 아래의 의미엔티티 모듈에 정의되어 있다. [[E1 "+" E2]]의 의미는 다음과 같이 약식으로 설명할 수 있다. 먼저 E1과 E2가 각각 평가되어 그 값 n_1과 n_2를 각각 내주고, 그 다음 그 값들의 합을 결과로 내준다. 이 약식 설명을 정식으로 정의된 위의 의미식 [2:]와 비교해보면 상당히 흡사함을 알 수 있다.

의미식 [1:]의 decimal은 디지트 문자열을 자연수로 바꾸는 데이타 연산이다. give Y는 함수양상 액션으로

*Y*가 산출한 값을 내준다. 액션결합자 and는 기본양상에 속하는 결합자로서 수행 순서는 구현 종속적이다. 의미식 [2:]와 [3:]을 보면 and의 두 부분액션 사이에 서로 간섭이 없기 때문에 수행의 순서는 중요하지 않다. and 결합자는 양쪽의 부분액션이 내준 값들의 튜플을 만들어 내준다. then 결합자는 왼쪽 부분액션이 내준 값을 오른쪽 부분액션으로 전달한다. given *d*는 산출자(yielder)로서 주어진 값이 *d* 종류(sort)임을 확인한 후 그 값을 산출한다. 여기에서 a, the, of 와 같은 표시들은 액션 표기의 판독성을 향상시키는 역할만 할 뿐 특별한 의미는 없다. check *Y*는 *Y*가 참값(true)을 산출하면 수행이 완료되고, *Y*가 거짓값(false)을 산출하면 수행이 실패한다. 의미식 [5:]는 액션결합자 or의 두 부분액션 중의 하나가 반드시 실패하게 되므로 결정적인 선택을 나타낸다.

의미함수 모듈에서 의미를 나타내는데 사용된 표기들은 아래의 의미엔티티 모듈에 명시되어 있다.

```
module: ExpCore/Semantic Entities.
  includes: Data Notation.
  introduces: value.
  (*) value = number | truth-value.
  (*) number =< integer.
endmodule: ExpCore/Semantic Entities.
```

위의 모듈에서 includes: Data Notation은 Peter Mosses의 책에 정의되어 있는 표준 데이타[3,7]가 이 모듈에 포함됨을 나타낸다.

3. 추상화 원칙의 적용

추상화 원칙에 따르면 구문 도메인에 속하면서 의미부여가 가능한 프로그램은 모두 이름을 부여할 수 있다[11]. 즉, ExpCore의 구문 도메인 Expr에 속한 표현식은 모두 이름을 부여할 수 있다는 말이 된다. 구문에 이름을 부여하는 과정을 바인딩(binding)이라고 하며, 바인딩을 통하여 환경이 생겨나게 된다. 부여된 이름은 추후에 호출하면 사용할 수 있다. 표현식 바인딩의 추상구문 모듈인 ExpBind는 아래와 같이 정의된다. 바인딩 구문은 선언(declaration)이라고 하는 새로운 구문 도메인 Decl에 모아 놓는다. Iden은 이름을 대표하는 새로운 구문 도메인이다.

```
module: ExpBind/Abstract Syntax.
  grammar:
  (*) Decl = [[ "val" Iden "=" Expr ]]
           | [[ Decl "," Decl ]].
  (*) Expr = Iden.
  (*) Iden = [[ letter (letter | digit)* ]].
  endgrammar.
endmodule: ExpBind/Abstract Syntax.
```

이름 호출의 의미는 표현식을 언제 평가하느냐에 따라서 달라질 수 있다. 표현식에 이름을 부여하여 바인딩을 만들기 전에 그 표현식을 평가할 수도 있고(적극적인 바인딩: eager binding), 그 이름을 호출한 후에 평가할 수도 있다(소극적인 바인딩: lazy binding).

3.1 적극적 바인딩의 의미

적극적 표현식 바인딩의 의미함수 모듈 ExpBindEager는 다음과 같이 정의할 수 있다.

```
module: ExpBindEager/Semantic Functions.
  needs: ExpBind/Abstract Syntax,
         ExpBindEager/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I:Iden; E:Expr; D1,D2:Decl;
  (*) elaborate _ :: Decl -> action[producing bindings].
  [7:] elaborate [[ "val" I "=" E ]] =
       evaluate E then bind I to the given number.
  [8:] elaborate [[ D1 "," D2 ]] =
       elaborate D1 and elaborate D2.
  (*) evaluate _ :: Expr -> action[giving a value]
                            [using current bindings].
  [9:] evaluate I = give the number bound to I.
endmodule: ExpBindEager/Semantic Functions.
```

의미함수 elaborate는 Decl에 속한 프로그램 트리를 바인딩을 생성하는 액션으로 매핑한다. 의미식 [7:]에 의하면 표현식 바인딩의 본체는 이름에 바인딩되기 전에 평가된다 (적극적인 평가). 이름의 부여가 가능한 값은 bindable이라고 하여 따로 구분하는데, 아래의 의미엔티티 모듈 ExpBindEager에서는 number 만이 bindable로 정의되어 있다. 즉, truth-value는 이름을 붙일 수 없다는 뜻이다. 의미식 [8:]의 액션결합자 and에 의하면 이름의 중복이 없는 두 바인딩은 독립적으로 생성된 후 결합된다. 이름 호출을 나타내는 I는 의미함수 [9:]에 명시된 대로 현재 바인딩을 참조하여 그 이름에 바인드된 값을 내준다. 다음 의미엔티티 모듈에서 token과 bindable을 정의한다.

```
module: ExpBindEager/Semantic Entities.
  includes: Data Notation, ExpCore/Semantic Entities.
  introduces: bindable, token.
  (*) bindable = number.
  (*) token = string.
endmodule: ExpBindEager/Semantic Entities.
```

3.2 소극적 바인딩의 의미

소극적 표현식 바인딩의 의미함수 모듈 ExpBindLazy는 다음과 같이 정의할 수 있다.

```
module: ExpBindLazy/Semantic Functions.
  needs: ExpBind/Abstract Syntax,
         ExpBindLazy/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I:Iden; E:Expr; D1,D2:Decl;
  (*) elaborate _ :: Decl -> action[producing bindings].
[10:] elaborate [[ "val" I "=" E ]] =
        bind I to the abstraction of evaluate E.
[11:] elaborate [[ D1 "," D2 ]] =
        elaborate D1 and elaborate D2.
  (*) evaluate _ :: Expr -> action[giving a value]
                              [using current bindings].
[12:] evaluate I = enact the abstraction bound to I.
endmodule: ExpBindLazy/Semantic Functions.
```

A를 액션이라고 할 때, 산출자 abstraction of A는 A를 캡슐화한 추상캡슐(abstraction)을 산출한다. 추상캡슐에 포함되어 있는 액션은 enact 액션에 의하여 활성화되어야 비로소 수행된다. 의미식 [10:]과 [12:]에 의하면, 표현식 E는 평가되지 않은 채 캡슐화되어 이름 I에 바인드되고, 이름 I가 호출되면 활성화되어 평가된다. 추상캡슐이 이름에 바인드되므로 아래의 의미엔티티 모듈 ExpBindLazy에서 bindable 값은 abstraction으로 정의된다.

```
module: ExpBindLazy/Semantic Entities.
  includes: Data Notation, ExpCore/Semantic Entities.
  introduces: bindable, token.
  (*) bindable = abstraction.
  (*) token = string.
endmodule: ExpBindLazy/Semantic Entities.
```

4. 제한 원칙의 적용

제한 원칙에 따르면 구문 도메인에 속하면서 의미 부여가 가능한 프로그램은 모두 지역선언을 허용할 수 있다 [11]. 즉, ExpCore의 Expr 구문도메인에 속하는 표현식은 모두 지역선언을 가질 수 있다. 지역선언을 가진 구문 구조를 블록(block)이라고 한다. 표현식 블록 ExpBlock의 추상구문 모듈과 의미함수 모듈은 다음과 같이 정의된다.

```
module: ExpBlock/Abstract Syntax.
  grammar:
  (*) Expr = [[ "let" Decl "in" Expr "end" ]].
  (*) Decl = .
  endgrammar.
endmodule: ExpBlock/Abstract Syntax.
module: ExpBlock/Semantic Functions.
  needs: ExpBlock/Abstract Syntax,
         (ExpBindEager|ExpBindLazy)/Semantic Entities.
  introduces: evaluate _.
  variables: D:Decl; E:Expr;
  (*) evaluate _ :: Expr -> action[giving a value]
                              [using current bindings].
[13:] evaluate [[ "let" D "in" E "end" ]] =
        furthermore elaborate D hence evaluate E.
endmodule: ExpBlock/Semantic Functions.
```

의미식 [13:]에서 furthermore elaborate D는 받은 바인딩을 elaborate D가 생성한 바인딩으로 덮어 씌운다. hence 결합자는 그 덮어 씌워진 바인딩을 evaluate E로 넘긴다. 이는 일반적인 블록 구조의 의미와 정확하게 일치한다. 따라서 D에서 선언된 바인딩은 본체 E에서만 참조할 수 있고, 그 밖에서는 참조가 불가능하다. ExpBlock 의미모듈이 ExpBindEager나 ExpBindLazy 의미엔티티 모듈 중 어떤 것도 사용가능하므로 | 연산자를 사용하여 표시한다.

블록 구조에는 영역결정을 동적으로(dynamic) 할 것인지 정적으로(static) 할 것인지에 대한 문제가 발생한다. 어떤 영역결정방법을 적용할 것인지는 "let" 블록의 의미식 [13:]에서는 표시할 수 없으므로, 3.1절에서 논의한 표현식 바인딩의 의미함수에서 표시한다. 3.1절에서 살펴 본 적극적 바인딩의 의미를 살펴보면, 바인딩을 생성할 당시에 제공되는 바인딩을 가지고 E를 평가하기 때문에 자연스럽게 정적으로 영역결정을 할 수밖에 없다. 반면에, 3.2절의 소극적 바인딩의 경우 추상캡슐이 활성화되면서 바인딩을 전혀 제공해주지 않기 때문에 문제가 있다. 의미식 [10:]과 [12:]를 살펴보면, 산출자 the abstraction of evaluate E는 evaluate E만을 포함하는 추상캡슐을 산출하고, 바인딩을 추상 캡슐에 포함시키지 않는다. 더구나, 활성화하는 액션 enact the abstraction bound to I를 수행할 때도 바인딩을 제공하지 않는다. 따라서, 정적이든지 동적이든지 영역결정규칙을 적용하기 위해서 바인딩을 제공해주는 closure Y 산출자가 필요하다 (여기서 Y는 산출자를 나타낸다). closure Y는 Y와 같은 추상캡슐을 산출하지만, 캡슐화한 추상캡슐에 그 당시의 바인딩을 포함시키고, 그 추상캡슐이 활성화될 때 포함되어 있는 바인딩을 가지고 액션을 수행한다.

4.1 정적영역결정의 의미

정적영역결정 방법을 사용하는 소극적인 표현식 바인

딩을 정의하는 의미함수 모듈은 다음과 같이 정의된다.

```
module: ExpBindLazyScopeStatic/Semantic Functions.
  needs: ExpBind/Abstract Syntax,
         ExpBindLazy/Semantic Entities.
  introduces: elaborate _, evaluate _ .
  variables: I:Iden; E:Expr;
  (*) elaborate _ :: Decl -> action[producing bindings].
[14:] elaborate [[ "val" I "=" E ]] =
        bind I to the closure abstraction of evaluate E.
[15:] elaborate [[ D1 "," D2 ]] =
        elaborate D1 and elaborate D2.
  (*) evaluate _ :: Expr -> action[giving a value]
                            [using current bindings].
[16:] evaluate I = enact the abstraction bound to I.
endmodule: ExpBindLazyScopeStatic/Semantic Functions.
```

의미식 [14:]에서 산출자 the closure abstraction of evaluate E는 액션 evaluate E와 그 시점의 바인딩을 포함하는 추상캡슐을 산출한다. 의미식 [16:]의 enact the abstraction bound to I 액션은 I에 바인드 된 추상캡슐을 활성화하는데, 이 때 수행되는 액션은 추상캡슐에 들어있는 바인딩을 제공받는다. 즉, 이름을 정의할 당시의 바인딩을 사용한다.

4.2 동적영역결정의 의미

동적영역결정 방법을 사용하는 소극적인 표현식 바인딩을 정의하는 의미함수 모듈은 다음과 같이 정의된다.

```
module: ExpBindLazyScopeDynamic/Semantic Functions.
  needs: ExpBind/Abstract Syntax,
         ExpBindLazy/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I:Iden; E:Expr;
  (*) elaborate _ :: Decl -> action[producing bindings].
[17:] elaborate [[ "val" I "=" E ]] =
        bind I to the abstraction of evaluate E.
[18:] elaborate [[ D1 "," D2 ]] =
        elaborate D1 and elaborate D2.
  (*) evaluate _ :: Expr -> action[giving a value]
                            [using current bindings].
[19:] evaluate I =
        enact the closure abstraction bound to I.
endmodule: ExpBindLazyScopeDynamic/
           Semantic Functions.
```

의미식 [17:]을 보면, I에 바인드되는 추상캡슐에는 그 당시의 바인딩을 포함시키지 않는다. 의미식 [19:]에서 산출자 the closure abstraction bound to I에서 I에 바인드된 추상캡슐에다 그 당시의 바인딩을 포함시키고, 활성화하면 포함된 바인딩을 가지고 액션을 수행한다. 즉, 호출할 당시의 바인딩을 사용하게 된다.

5. 파라미터화 원칙의 적용

파라미터화 원칙에 따르면 구문도메인에 속하면서 의미 부여가 가능한 프로그램은 모두 파라미터가 될 수 있다 [11]. 즉, ExpCore의 Expr 구문 도메인에 속하는 표현식은 모두 파라미터가 될 수 있다. 표현식 파라미터를 가진 표현식 바인딩의 추상구문은 다음과 같이 정의한다.

```
module: ExpParam/Abstract Syntax.
  grammar:
  (*) Decl = [[ "fun" Iden "(" Iden ")" "=" Expr ]].
  (*) Expr = [[ Iden "(" Expr ")" ]]
           | Iden.
  endgrammar.
endmodule: ExpParam/Abstract Syntax.
```

파라미터와 함께 호출을 하는 경우 실제 파라미터(인수)를 언제 평가하느냐에 따라서 의미가 달라질 수 있다. 이 절에서는 값호출(call-by-value)과 이름호출(call-by-name) 파라미터 전달방법의 의미구조에 대해서 알아본다.

5.1 값호출 파라미터 전달

값호출 파라미터 전달 방식을 사용하여 호출을 하면, 그 인수를 평가한 뒤 평가된 결과 값이 형식파라미터에 바인드된다. 의미함수 모듈은 다음과 같이 정의할 수 있다.

```
module: ExpParamCBV/Semantic Functions.
  needs: ExpParam/Abstract Syntax,
         ExpBindEager/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I,I1,I2:Iden; E:Expr;
  (*) elaborate _ :: Decl -> action[producing bindings].
[20:] elaborate [[ "fun" I1 "(" I2 ")" "=" E ]] =
        bind I1 to the closure abstraction of
          (furthermore bind I2 to the given number
           hence evaluate E).
  (*) evaluate _ :: Expr -> action[giving a value]
                            [using current bindings].
[21:] evaluate [[ I "(" E ")" ]] =
        (give the abstraction bound to I
         and
         evaluate E)
```

```
        then
            enact the application of the given abstraction#1
                                    to the given number#2.
    [22:] evaluate I = give the number bound to I.
    endmodule: ExpParamCBV/Semantic Functions.
```

의미식 [20:]에 의하면, "주어진 수를 형식파라미터 I2에 바인드하고, 그 바인딩을 받은 바인딩에 덮어씌운 뒤, E를 평가하는 액션"을 추상캡슐화한 뒤 I1에 바인드한다.

의미식 [21:]에 따르면, 인수인 E를 먼저 평가한 후 그 결과를 I에 바인드된 추상캡슐을 활성화시키면서 제공한다. 제공된 값은 형식 파라미터에 바인드되며, 그 바인딩은 본체 E에서 지역선언 역할을 한다. 의미함수 [20:]을 보면 정적영역결정 규칙이 적용됨을 알 수 있다. ExpBindEager의 의미엔티티 모듈이 그대로 사용된 것을 보면, 적극적 바인딩과 값호출 파라미터 호출 방식이 의미적으로 유사하다는 사실을 알 수 있다.

5.2 이름호출 파라미터 전달

이름호출 파라미터 전달 방식에서 호출된 본체에서 사용될 때까지 인수의 평가가 연기된다. 즉, 평가되지 않은 인수는 형식 파라미터에 바인드되고, 형식파라미터가 본체에서 참조될 때마다 평가된다. 의미함수 모듈은 다음과 같이 정의할 수 있다.

```
module: ExpParamCBN/Semantic Functions.
  needs: ExpParam/Abstract Syntax,
         ExpBindLazy/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I,I1,I2:Iden; E:Expr;
  (*) elaborate _ :: Decl -> action[producing bindings].
[23:] elaborate [[ "fun" I1 "(" I2 ")" "=" E ]] =
        bind I1 to the closure abstraction of
          (furthermore
             bind I2 to the given abstraction
           hence evaluate E).
  (*) evaluate _ :: Expr -> action[giving a value]
                                [using current bindings].
[24:] evaluate [[ I "(" E ")" ]] =
        enact the application
                of the abstraction bound to I
                to the closure abstraction of evaluate E.
[25:] evaluate I = enact the abstraction bound to I.
endmodule: ExpParamCBN/Semantic Functions.
```

의미식 [23:]과 [24:]를 보면 함수가 적용될 때 인수 E는 평가되지 않고, 해당 액션 *evaluate* E가 추상캡슐로 캡슐화되어 전달되어, 형식파라미터에 바인드된다. 의미식 [25:]에서 보여주듯이, 전달된 추상캡슐은 파라미터화된 추상의 본체에서 호출되면 활성화된다. 의미함수 [24:]에서 closure는 그 시점의 바인딩을 추상캡슐에 포함시키므로, 호출된 후 인수가 평가될 때 포함된 바인딩이 제공된다. 이는 정적 영역결정방식을 사용함을 의미한다. 여기서도 ExpBindLazy의 의미 엔티티 모듈이 그대로 사용된 것을 보면, 소극적 바인딩과 이름호출 파라미터 호출 방식이 의미적으로 서로 유사하다는 사실을 알 수 있다.

5.3 재귀바인딩

추상의 본체에서 바인딩하는 이름의 참조가 가능하게 하기 위해서는 bind 액션을 recursively bind로 바꾸기만 하면 된다. 이는 순환구조를 만들어 재귀참조를 가능하게 한다. 액션 recursively bind I to Y에서는 Y에서 나타나는 모든 I 참조는 I에 바인드된 추상캡슐을 가리킨다. 다음은 파라미터화된 재귀 표현식 재귀바인딩과 관련한 의미함수 모듈이다.

```
module: ExpParamCBVRec/Semantic Functions.
  needs: ExpParam/Abstract Syntax,
         ExpBindEager/Semantic Entities.
  introduces: elaborate _, evaluate _.
  variables: I,I1,I2:Iden; E:Expr;
  (*) elaborate _ :: Decl -> action[producing bindings].
[26:] elaborate [[ "fun" I1 "(" I2 ")" "=" E ]] =
        recursively bind I1 to
          the closure abstraction of
            (furthermore bind I2 to the given number
             hence evaluate E).
  (*) evaluate _ :: Expr -> action[giving a value]
                                [using current bindings].
[27:] evaluate [[ I "(" E ")" ]] =
        (give the abstraction bound to I
         and
         evaluate E)
        then
         enact the application of the given abstraction#1
                                to the given number#2.
[28:] evaluate I = give the number bound to I.
endmodule: ExpParamCBVRec/Semantic Functions.
```

6. 일차 함수형 언어 모듈의 구성

지금까지 정의한 추상구문 모듈, 의미함수 모듈, 의미

엔티티 모듈은 일차 함수형 언어(first-order functional language)를 정의하는데 사용할 수 있다. 정적 영역결정 규칙을 준수하는 적극적 일차 함수형 언어는 다음과 같이 정의할 수 있다.

```
module: FirstOrderEagerFunctional/Action Semantics.
  includes: ExpCore/Action Semantics,
            ExpBind/Abstract Syntax,
            ExpBlock/Abstract Syntax,
            ExpParam/Abstract Syntax,
            ExpBindEager/Semantic Function,
            ExpBlock/Semantic Function,
            ExpParamCBVRec/Semantic Function,
            ExpBindEager/Semantic Entities.
endmodule: FirstOrderEagerFunctional/Action Semantics.
```

위의 언어 모듈은 ExpCore/ActionSemantics 모듈을 그대로 포함하고 있으며, 적극적인 평가와 관련된 확장 모듈들을 포함하여 정의한다. 구문으로는 ExpBind와 ExpBlock과 ExpParam의 추상구문 모듈들이 추가되고, 적극적 바인딩과 값호출 파라미터 전달 방법을 채택한 의미함수를 포함하여 적극적으로 평가하는 언어를 정의하였다. 이 언어 모듈에 포함된 모든 정의는 전혀 정의상의 충돌이 발생(같은 구문이 다른 의미를 가지는 경우 충돌이 발생한다고 함)하지 않기 때문에 획일성과 직교성 원칙을 만족한다고 할 수 있다.

정적 영역결정 규칙을 준수하는 소극적 일차 함수형 언어도 비슷한 요령으로 소극적 바인딩과 이름호출 파라미터 전달방법을 채택한 모듈들을 포함시켜서 다음과 같이 정의할 수 있다.

```
module: FirstOrderLazyFunctional/Action Semantics.
  includes: ExpCore/Abstract Syntax,
            ExpBind/Abstract Syntax,
            ExpBlock/Abstract Syntax,
            ExpParam/Abstract Syntax,
            ExpCore/Semantic Function,
            ExpBindLazyScopeStatic/Semantic Function,
            ExpBlock/Semantic Function,
            ExpParamCBN/Semantic Function,
            ExpBindLazy/Semantic Entities.
endmodule: FirstOrderLazyFunctional/Action Semantics.
```

앞에서 정의한 두 언어모듈은 포함된 모듈의 정의들 간에 충돌이 없도록 잘 조화시켜 정의하였다. 그러나, 서로 상반된 의미를 가지는 모듈들을 포함시키는 경우 일관성이나 직교성과 같은 언어 설계 원칙에 위배되는 경우가 발생한다. 예로써, 다음 모듈을 살펴보자.

```
module: BadFunctional/Action Semantics.
  includes: ExpCore/Action Semantics,
            ExpBind/Abstract Syntax,
            ExpBlock/Abstract Syntax,
            ExpParam/Abstract Syntax,
            ExpBindEager/Semantic Function,
            ExpBlock/Semantic Function,
            ExpParamCBN/Semantic Function,
            ExpBindEager/Semantic Entities,
            ExpBindLazy/Semantic Entities.
endmodule: BadFunctional/Action Semantics.
```

유일하게 정의의 충돌이 일어나는 부분은 다음 두 식이다.

```
[9:] evaluate I = give the number bound to I.
[25:] evaluate I = enact the abstraction bound to I.
```

이 두 식은 완전히 다르기 때문에 이 두 식을 다 수용하는 방법은 or 결합자를 사용하여 다음과 같이 정의하는 수밖에 없다.

```
evaluate I = give the number bound to I.
               or
             enact the abstraction bound to I.
```

이 식에 의하면, I에 바인드된 값이 number 종류이면 바인드된 수를 내주고, abstraction 종류이면 그 추상캡슐을 활성화한다. 물론 이 식은 합법적인 액션 의미표기이다. 그러나, 같은 구조가 두 가지 다른 의미를 가지게 되므로, 언어설계의 획일성 원칙에 위배된다. 이러한 결과가 발생한 이유는 다른 성질을 가진 두 개의 모듈(적극적인 바인딩과 값호출 파라미터 전달)을 같은 모듈에 포함 시켰기 때문이다. 이는 일반적으로 바람직하지 않은 설계라고 여겨진다.

7. 고차 표현식 확장

지금까지 정의한 언어모듈에서는 1차 값(first-order value)들만 표현할 수 있었다. 고차 프로그래밍(higher-order programming)을 하기 위해서는 함수도 표현식으로 나타낼 수 있어야 한다. 이 절에서는 함수를 표현식으로 나타낼 수 있는 람다계산표기법(λ-calculus)의 의미모듈을 정의한다. 람다계산표기법의 추상구문 모듈은 다음과 같다.

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

```
(*) Iden = [[ letter (letter | digit)* ]].
endgrammar.
endmodule: LambdaCalculus/Abstract Syntax.
```

7.1 이름호출 람다계산표기법

이름호출 람다계산표기법(call-by-name λ-calculus)의 의미함수 모듈은 이름호출 파라미터 전달방법을 쓰기 때문에 의미함수가 ExpParamCBN 모듈과 매우 흡사하다.

```
module: LambdaCalculusCBN/Semantic Functions.
  needs: LambdaCalculus/Abstract Syntax.
         LambdaCalculusCBN/Semantic Entities.
  introduces: evaluate _.
  variables: I:Iden; E,E1,E2:Expr;
  (*) evaluate _ :: Expr -> action[giving a value].
[29:] evaluate [[ "lam" "(" I ")" "=>" E ]] =
        give the closure abstraction of
         (furthermore bind I to the given bindable
          hence evaluate E).
[30:] evaluate [[ E1 E2 ]] =
        evaluate E1 then
        enact the application of the given abstraction
              to the closure abstraction of evaluate E2.
[31:] evaluate I = enact the bindable bound to I.
endmodule: LambdaCalculusCBN/Semantic Functions.
```

의미식 [29:]에서 볼 수 있듯이 함수의 의미는 액션표기법으로 추상캡슐(abstraction)로 표현된다. 여기에 추상캡슐은 일시 값으로 취급되기 때문에, value에 포함되도록 아래의 의미엔티티 모듈에서 정의한다.

```
module: LambdaCalculusCBN/Semantic Entities.
  includes: Data Notation,
            ExpBindLazy/Semantic Entities.
  introduces: value.
  (*) value = ... | abstraction.
endmodule: LambdaCalculusCBN/Semantic Entities.
```

7.2 값호출 람다계산표기법

값호출 람다계산표기법(call-by-value λ-calculus)의 의미함수 모듈은 값호출 파라미터 전달방법을 쓰기 때문에 의미함수가 ExpParamCBV 모듈과 매우 흡사하다.

```
module: LambdaCalculusCBV/Semantic Functions.
  needs: LambdaCalculus/Abstract Syntax.
         LambdaCalculusCBV/Semantic Entities.
  introduces: evaluate _.
  variables: I:Iden; E,E1,E2:Expr;
  (*) evaluate _ :: Expr -> action[giving a value].
[32:] evaluate [[ "lam" "(" I ")" "=>" E ]] =
        give the closure abstraction of
         (furthermore bind I to the given bindable
          hence evaluate E).
[33:] evaluate [[ E1 E2 ]] =
        (evaluate E1 and evaluate E2) then
         enact the application of the given abstraction#1
                to the given bindable#2.
[34:] evaluate I = give the bindable bound to I.
endmodule: LambdaCalculusCBV/Semantic Functions.
```

실제 파라미터는 함수 호출시 계산되기 때문에 abstraction(함수를 나타냄)과 number가 bindable에 속한다.

```
module: LambdaCalculusCBV/Semantic Entities.
  includes: Data Notation,
            ExpBindEager/Semantic Entities.
  introduces: value, bindable.
  (*) value = ... | abstraction.
  (*) bindable = ... | abstraction.
endmodule: LambdaCalculusCBV/Semantic Entities.
```

7.3 고차 함수형 언어의 구성

적극적인 고차 함수형 언어는 다음과 같이 정의할 수 있다.

```
module: HigherOrderEagerFunctional/Action Semantics.
  includes: FirstOrderEagerFunctional/Action Semantics,
            LambdaCalculus/Abstract Syntax,
            LambdaCalculusCBV/Semantic Function,
            LambdaCalculusCBV/Semantic Entities.
endmodule: HigherOrderEagerFunctional/Action Semantics.
```

이 새로운 언어 모듈은 FirstOrderEagerFunctional 언어 모듈을 확장한 것으로, 값호출 람다계산표기법과 관련된 모듈들을 포함하여 구성된다. 확장된 언어 모듈이 획일성과 직교성 원칙을 위배하지 않는지 검사해보는 건 별로 힘들지 않다. 포함된 언어 모듈에서 두 번 이상 나타난 언어 구조가 하나 이상의 다른 의미를 가지게 되면, 일단 의심해 보아야 한다. 이 확장 모듈에서는 유일하게 I가 한 번 이상 정의되어 있다. 포함된 모듈에서 I의 의미들을 모아보면 다음과 같다.

```
[9:] evaluate I = give the number bound to I.
[28:] evaluate I = give the number bound to I.
[34:] evaluate I = give the bindable bound to I.
```

확장된 언어 모듈이 획일성과 직교성 원칙을 만족하기 위해서 위의 의미 함수는 모두 동일해야 한다. 언뜻 보기에 처음 두 식은 동일하지만 마지막 식은 달라 보

인다. 의미식 [9:]와 [28:]은 둘 다 의미엔티티 모듈 ExpBindEager를 사용하는 반면, 의미식 [34:]는 LambdaCalculusCBV를 사용함을 주목해 보자. 그러면, 앞의 모듈에 number가 bindable로 정의되어 있고, 뒤의 모듈은 앞의 모듈을 포함하고 있으므로, 의미는 모두 동일함을 알 수 있다. 따라서 합성된 언어 모듈에서 의미식 [34:]을 선택해도 무방하다.

소극적 고차 함수형 언어도 비슷하게 다음과 같이 정의할 수 있다.

```
module: LazyFunctional/Action Semantics.
  includes: FirstOrderLazyFunctional/Action Semantics,
            LambdaCalculus/Abstract Syntax,
            LambdaCalculusCBN/Semantic Function,
            LambdaCalculusCBN/Semantic Entities.
endmodule: LazyFunctional/Action Semantics.
```

8. 결 론

이 논문에서는 액션 의미표기법에 기반한 언어 정의를 모듈화 하여, 새로운 언어를 설계하는 과정에서 체계적으로 언어 정의를 확장하고 합성하는 방법을 제시하였다. 액션 의미표기법은 모듈성과 확장성이 좋으므로, 언어 모듈들을 합성하여 새로운 언어 모듈을 구성하더라도 의미 정의 상에 충돌이 없으면 기존의 모듈들을 거의 수정할 필요가 없었다.

이 논문에서 제시한 액션 의미표기법에 기반한 언어 모듈 구성 및 확장 방법은 특정도메인용(domain-specific) 언어나 신속하게 실험개발될 특수목적용(rapidly prototyped special-purpose) 언어를 설계하는데 유용한 안내 도구로서 사용될 수 있다. 즉, 특정 응용분야에 해당되는 연산을 포함한 언어 핵심 모듈과 확장 모듈을 설계하고 합성하여 특정도메인용 언어를 설계할 수 있다.

추후과제로 실질적인 복합 파라다임(multi-paradigm) 언어와 특정도메인용(domain-specific) 언어를 설계하는데 사용될 수 있는 다양한 언어모듈 라이브러리의 작성을 생각해 볼 수 있다.

참 고 문 헌

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

[2] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, 1986.

[3] Peter D. Mosses, *Action Semantics*, Cambridge Tracts in Theoretical Computer Science 26, Cambridge University Press, 1992.

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

[5] 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* 1113, 37-61, Springer-Verlag, 1996.

[6] Peter D. Mosses, A tutorial on action semantics, Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996), March 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, March 1993.

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

[10] David A. Watt, The static and dynamic semantics of Standard ML, In *IWAS '99, 2nd International Workshop on Action Semantics (ed. Mosses, P.D., and Watt, D.A.)*, BRICS NS-99-3, 155-172, University of Aarhus, 1999.

[11] David A. Schmidt, *The Structure of Typed Programming Languages*, MIT Press, 1994.

[12] Kyung-Goo Doh and David A. Schmidt, Extraction of strong typing laws from action semantics definitions, In *ESOP'92, Proc. European Symposium on Programming, Rennes, Lecture Notes in Computer Science* 582, 151-166, Springer-Verlag, 1992.

[13] Kyung-Goo Doh and David A. Schmidt, Action semantics-directed prototyping, *Computer Languages*, 19(4): 213-233, Pergamon Press, 1993.

도 경 구

1980년 한양대학교 공학사. 1987년 미국 Iowa State University 전산학석사. 1992년 미국 Kansas State University 전산학박사. 1993년 4월 ~ 1995년 9월 일본 University of Aizu 교수. 1995년 9월 ~ 현재 한양대학교 교수. 관심분야는 프로그래밍언어, 소프트웨어설계, 스마트카드, 전자상거래 기반기술