객체 지향 언어를 위한 의미 명세

Specification of Semantics for Object Oriented Programming Language

한 정 란*
Jung Lan Han

요 약

의미 기반 표기법은 새로운 프로그래밍 언어를 명세하여 그 언어를 구현하기 위한 정적이고 동적인 의미를 명세하는 데 필요한 것이다. 프로그래밍 언어에 대한 의미를 보다 실제적으로 정의하면 그 의미에 따라 언어를 구현함으로써 번역기를 쉽게 만들 수 있다. 본 논문에서는 번역기를 쉽게 만들 수 있는 정적이고 동적인 의미를 명세하고자 한다.

객체 지향 언어를 위한 정적이고 동적인 의미를 적절하게 명세할 수 있도록 속성 문법을 확장하고 변형하여 실제적인 의미 기반의 작용식을 제시하였고 기존의 연구 방법과 비교하여 제시된 작용식의 의미 명세가 우월함을 알 수 있었다.

Abstract

Semantics-based notations need to be used for specification of static and dynamic semantics to specify and implement new programming language. If the semantics is practically defined, we easily gain a translator according to the implementation of the semantics. In this paper, we describe the static and dynamic semantics to get a translator easily.

We present practical semantics-based Action Equations, an extension and transformation of Attribute Grammar(AGs) suitable for specifying the static and dynamic semantics of a object oriented programming language. Compare with the existing descriptions, Action Equations is superior, modernized, and accurate.

☞ keywords : Semantics, Dynamic Semantics, Operational Semantics, Attribute Grammar, Action Equations

1. 서 론

프로그래밍 언어를 설계하고 구현하는 것은 전자계산학에서 중요한 주제이다. 새로운 프로그래밍 언어를 정의하려면 언어의 구문과 의미를 정확하게 명세해야 한다. 언어 구문의 경우, 정규문법이나 문맥 자유 문법 등을 통해 표현하고, 의미 명세의 경우는 실제적으로 언어를 구현할 수 있도록 구체적이고 정확하게 명세하는 것이 중요하고 언어의 정적이고 동적인 의미를 명세해야 한다.

속성 문법은 언어의 정적 의미를 표현하는 형식적인 표기법으로서 동적 의미를 표현하기에는 부적절하다. 동적 의미를 잘 명세하고 명세된 언어를 구현하기 위해서는 기존의 속성 문법을 확장하고 변형하여 언어 구현에 필요한 동적인 작용들(actions)을 명세해야 할 필요가 있다. 이를 위해 속성 문법을 확장하고 변형하여 정적이고 동적인 의미를 쉽게 명세할 수 있는 새로운 작용식(Action Equations)[1]을 제시하는데 객체 지향 특성을 다룰 수 있는 작용식을 정의한다. 본 논문에서는 객체 지향 언어인 새로운 OBLA(Object Language) 언어를 EBNF(Extended Backus Naur Form) 표기법으로 정의하고 이 언어의 객체 지향 특성을 처리하는 작용식을 구체적이고 명확하게 명세한다.

판독성(Readability), 지능성(Intelligibility), 모듈성(Modularity), 확장성(Extensibility), 응용성(Applicability)의 다섯 영역에서 명세된 작용식을 기존의 의미 표현법과 비교하여 본 작용식의 우월성을 확인하고자 한다.

* 종신회원 : 협성대학교 경영학부 경영정보 교수
jlhan@uhs.ac.kr
[2007/07/04 투고 - 2007/07/12 심사 - 2007/07/30 심사완료]

2. 관련 연구

객체 지향 언어의 의미를 정의한 논문들 중에서 자바에 대한 의미 표현을 정의한 연구들[2, 5]을 살펴보면 크게 세 가지 표현법으로 분류할 수 있다.

- Alves-Foss와 Lam의 Denotational Semantics(DS) 표현[5]
- Borger and Schulte의 Abstract State Machine(ASM) 표현[5]
- Watt 와 Brown의 Action Semantics(AS) 표현[2]

DS(Denotational Semantics) 표현은 보다 통상적인 λ-표기법을 사용하고 연속 전달(continuation passing) 방식으로 작성되어 있다. 이러한 전달 방식을 사용해 각 구문 구조의 의미가 고 차원(Higher-order) 함수에 의해 표현되어진다. 연속(continuation)은 이러한 명령 함수의 인수들 중의 하나이고 프로그램의 나머지 동작들을 표현하는데 사용되는 함수이다. 자바에 대한 DS 표현은 오류가 많고 잘 정의되어있지 않아 자바를 완전하게 표현하기 힘들다[2].

ASM(Abstract State Machine) 표현은 부 표현(sub description)의 시리즈로 구성되고 고도의 단순화된 추상 구문이다. 예로서, For 나 Do 문장은 동등한 While 구문으로 나타내고 Switch 구문은 동등한 If 문으로 나타낸다. ASM 표현은 추상 구문으로 자바 그 자체로 나타나지 않는다. 예로서, 메소드 호출식은 호출된 메소드의 종류와 서명(signature)에 관한 정보로 확장되어진다. ASM 표현은 상태들 사이에 전이를 통해 상태 공간을 정의함으로써 기본적으로 작동하고 이 상태는 복잡한 구조를 갖는다. ASM 표현은 자바의 전체 구문으로 확장 가능하지만 많은 구조를 처리하기에는 불완전한 측면이 있다[2]. ASM은 자바의 실제 구문과는 다른 추상구문으로 매우 명확하지만 저급 수준이고, 제한된 의미에서만 모듈방식으로 사용된다[2].

AS(Action Semantics) 표현은 스레드와 메소드 중복(overloading)을 제외한 모든 자바 언어를 표현할 수 있고 모듈의 집합으로 구조화되어 있다. 상위 수준의 모듈인 의미 함수들은 특별한 구문들인 문장, 수식, 선언문에 대한 의미 방정식(semantics equations)을 포함하는 각각의 부 모듈을 포함한다. 의미 개체(entities) 모듈은 서로 다른 종류의 자료와 그 자료와 연관된 연산 즉 기본적인 자료, 값, 변수, 자료형, 클래스, 객체 등을 명세하는 부 모듈을 포함한다. AS 표현은 DS의 모듈화를 개선한 것으로 동작(action)이라 불리는 표시(denotations) 형식을 취하고 있고 동작은 여러 가지 다양한 기본적인 것(primitive)과 연결자(combinator)로 구성된 고정된 동작 개념을 사용해 표현되어진다. AS 표현은 프로그래밍 언어를 구현할 수 있도록 구체적으로 표현되어 있지 않아 실제로 번역기를 만들 때 어려움이 있다.

본 연구에서는 자바라는 특정한 언어가 아닌 일반적인 객체 지향 언어인 OBLA(OBject LAnguage)를 정의하여 이 언어에 대한 의미를 작용식(Action Equations)[1]을 통해 표현하고, 다른 기존의 연구들보다 구체적이고 확장 가능하고 응용력이 뛰어나서 번역기를 쉽게 구현할 수 있도록 동적 의미를 표현하고자 한다.

3. OBLA(OBject LAnguage) 언어

어떤 언어에 대한 번역기를 구현하기 위해서는 언어의 동적 의미 구조를 잘 표현하는 것이 중요하다. 따라서 본 논문에서는 속성 문법을 확장하고 변형하여 정적이고 동적인 의미를 표현하는 작용식(Action Equations)을 제시하고, 작용식에 들어가는 구문(syntax)을 정의하기 위해 (그림 1)에 OBLA(OBject LAnguage)를 EBNF로 표시하여 각 명령문의 구문을 정의한다.

<program>::=<class_list>

<class_list>::=<class_module>{<class_module>}

```
<class_module>::=[public] class
                <class_name>[<derived_class>]<cla
                ss_body>
<derived_class>::= : parent <class_name>
<class_body>::=<start_sym>[<declaration_part>]<m
                ethod_decl><end_sym>
<declaration_part>::=<declaration>{<declaration>}
<declaration>::=<qualifier_list><identifier_list>
                |<class_id_list>
<class_id_list>::=<class_decl>{<class_decl>}
<class_decl>::=<class_id><object_id>
<qualifier_list>::=<modifier>[<access>]
<modifier>::=public | private | protected
<identifier_list>::=<identifier>{,<identifier>}
<access>::=static | final
<method_decl>::=[<method_part>]<start_method>
<start_method>::= public static main (
                <identifier_list> ) <method_body>
<method_part>::=<method>{,<method>}
<method>::=<qualifier_list><method_heading><met
                hod_body>
<method_heading>::=<method_name><parameter_list>
<parameter_list>::=(<var_list>)
<method_body>::=<start_sym>[<declaration_part>]<
                method_stmt><end_sym>
<method_stmt>::=<stmt_list>[<return_statement>]
<stmt_list>::=<statement>{<statement>}
<statement>::=<in_statement>
                |<out_statement>
                |<ass_statement>
                |<if_statement>
                |<for_statement>
                |<while_statement>
                |<call_statement>
                |<new_statement>
<in_statement>::=read <var_list>
<out_statement>::=write <out_list>
<out_list>::=<exp_title_list>{,<exp_title_list>}
<ass_statement>::=<env_id> = <exp_list>
<exp_title_list>::=<title_list> | <exp_list>
<title_list>::="<title>"
<exp_list>::=<exp> | <bool>
<if_statement>::=if <condition><then_part>
                [<else_part>]
<then_part>::=then <com_statement_list>
<else_part>::=else <com_statement_list>
<com_statement_list>::=<start_sym><statement_list>
                <end_sym>
<for_statement>::=for <identifier> = <exp> to
                <exp> <com_statement_list>
<while_statement>::=while <condition>
                <com_statement_list>
<condition>::=<exp><rel_op><exp>
<call_statement>::=[<env_id>]<call_name>(<actual_
                list>)
<env_id>::=<class_obj_name>.
<class_obj_name>::=<class_id>|<object_id>
<return_statement>::=return <exp_list>
<actual_list>::=<actual_parm>{,<actual_parm>}
<new_statement>::=[<class_id>]<object_id> = new
                <class_id>(<actual_list>)
<start_sym>::= {
<end_sym>::= }
```

(그림 1) OBLA 언어 문법

OBLA 언어는 객체 지향 언어로 일반적 언어의 명령문들인 배정문, if 문, while 문, for 문 및 입출력문을 다루고 있다. 객체(object)를 표현하기 위해 사용자가 정의한 자료형이 클래스이고 새로운 클래스를 정의할 때는 객체를 나타내는데 필요한 자료 부분과 객체에 적용할 수 있는 연산 부분을 정의한다.

OBLA 언어의 클래스는 공용(public) 파트, 전용(private) 파트, 보호(protected) 파트로 구성되어진다. 전용 파트는 사용자가 이 자료를 직접 이용할 수 없고 클래스 내에 선언된 멤버 함수를 통해

사용할 수 있고 보호 파트는 선언된 클래스와 하위 클래스에서 자료를 이용할 수 있다. 공용 파트의 자료를 참조하기 위해서는 다음과 같이 작성한다.

class_name.id

예를 들면 **date** 라는 객체의 변수 **x** 를 참조하기 위해 **date.x** 라고 표현한다.

4. 작용 식

어떤 언어에 대한 번역기를 구현하기 위해서는 언어의 동적 의미 구조를 잘 표현하는 것이 중요하다. 따라서 본 논문에서는 속성 문법을 확장하고 변형하여 정적이고 동적 의미 구조를 표현하는 작용식(Action Equations)을 제시하고자 한다.

작용식에서 사용하는 각각의 속성들을 설명하면 다음과 같다.

- **out** 속성: 각 작용식을 실행한 후에 발생하는 결과를 나타내는 합성(synthesized) 속성
- **val** 속성: 각 비단말(nonterminal)의 값을 나타내는 합성 속성
- **env** 속성: 비단말의 환경을 나타내는 것으로 메소드를 호출하는 환경과 호출되는 환경을 구분하기 위한 전이(inherited) 속성
- **pri** 속성: 수식에서 연산자의 우선 순위를 지정하기 위한 속성
- **name** 속성: 식별자의 이름을 나타내는 속성
- **addr** 속성: 메소드를 호출하기 위해 주소를 저장하는 속성
- **type** 속성: 식별자의 형을 명시하는 속성
- **penv** 속성: 하위 클래스에서 상위 클래스의 환경을 물려받기 위해 상위 클래스의 환경을 저장하는 속성

각 작용식(action equation)에서 필요한 함수나 절차적인 수행을 필요로 하는 모듈에는 lookup, print_out, repeat, make_table, return_save, control_transfer 모듈이 있다.

- **lookup** 모듈: 변수 이름을 받아서 그 변수에 저장된 값을 반환하는 함수
- **print_out** 모듈: 변수나 수식의 값을 화면에 출력하는 함수
- **repeat** 모듈: 작용식에서 반복적으로 수행되는 작용을 표현하는 모듈로 repeat_start가 나오면 처음부터 다시 수행하고 repeat_end가 나오면 반복 수행을 끝낸다.
- **make_table** 모듈: 메소드에서 사용된 형식 매개변수를 심블 테이블에 만들기 위한 모듈이다. 각 매개변수의 이름과 메소드의 이름과 환경을 사용해서 매개변수에 대한 정보를 구성한다. 각 메소드에 대해 한번만 실행되는 모듈이다.
- **return_save** 모듈: 메소드를 호출할 때 실행하는 모듈로 호출되는 메소드의 반환 주소를 저장한다.
- **control_transfer** 모듈: 호출하거나 반환될 때 프로그램의 제어를 이동시키는 모듈이다.
- **determine_environment** 모듈: 식별자 이름이 주어졌을 때 그 식별자의 환경을 반환하는 모듈이다. 식별자가 클래스에 속한 경우 클래스 이름을 반환하고 메소드 안에 선언된 경우 메소드 이름을 반환한다.
- **search_member** 모듈: 클래스 이름이 주어졌을 때 그 클래스안에 선언된 멤버 변수들을 반환하는 모듈이다.

메소드의 경우 메소드에서 사용된 각각의 형식 매개변수들의 환경과 이름을 심블 테이블에 등록한다. 환경은 env 속성을 사용하여 저장하고 메소드의 이름으로 환경 속성을 표현한다.

작용식 중에서 Execute equation에 대해 객체 지향 언어를 다루기 위한 동적 의미를 명세하고 나머지 작용식은 참고문헌[1]에 자세히 기술되어 있다.

• Execute equation

Execute equation은 실행 후에 'diverge'하거나 'complete'하는 사건이 발생할 수 있다. Execute equation은 각 명령문을 실행하는 동적 명세를 표현하고 순차적으로 실행되는 절차적인 작용(action)이다. 각 식에서 s1, s2, s는 명령문을 나타내고 <>는 비단말을 의미한다.

Execute equation에서는 배정문, if 문, while 문, for문, 입출력문, 메소드 처리문과 클래스의 동적 의미 구조를 표현한다. **class 객체 선언문**은 심블 테이블에서 해당되는 클래스 이름을 찾아 새로 선언된 객체를 심블 테이블에 등록하고 객체의 클래스의 환경을 찾은 클래스 명으로 지정한다. 객체를 사용하기 위해 해당하는 자료에 메모리를 할당한다. **class 문**은 클래스의 시작 주소를 addr 속성에 저장하고 클래스의 환경을 클래스 이름으로 정한다. 하위 클래스일 경우는 penv 속성에 상위 클래스의 환경(env)을 저장하고 클래스 이름을 심블 테이블에 만든다.

◀Execute [stmts] → **event** ⟦ complete | diverge ⟧

· Execute [<s_1 s_2>] →
  s_1.out ← Execute [<s_1>]
  s_2.out ← Execute [<s_2>]

· Execute [**if** <con> **then** <s_1>, <s_2>, ... <s_n> where n ≥1] →
  con.val ← Eval_rel [<con>]
  if con.val then
    for i=1 to n do
      Execute [<s_i>]
    od
  endif

· Execute [**if** <con> **then** <s_1>, <s_2>, ... <s_n> **else** <s_1>, <s_2>, ... <s_n>where n ≥1] →
  con.val ← Eval_rel [<con>]
  if con.val then
    for i=1 to n do
      Execute [<s_i>]
    od
  else
    for j=1 to n do
      Execute [<s_j>]
    od
  endif

· Execute [**while** <con> <s_1>, <s_2>, ... <s_n>where n ≥1] →
  repeat
    con.val ← Eval_rel [<con>]
    if con.val then
      for i=1 to n do
        Execute [<s_i>]
        repeat_start
      od
    endif
  repeat_end

· Execute [**for** <id> = <exp_1> **to** <exp_2> <s_1>, <s_2>, ... <s_n>where n ≥1] →
  id.val ← Evaluate [<exp_1>]
  exp_2.val ← Evaluate [<exp_2>]
  repeat
    if exp_2.val is greater than and equal to id.val
      then con.val ← true
      else con.val ← false
    endif
    if con.val then
      for i=1 to n do
        Execute [<s_i>]
      od
      id.val ← id.val + 1
      repeat_start

endif
repeat_end

· Execute [<method_name>(<$param_1$>, <$param_2$>, ..., <$param_n$>) where n ≥1] →
for i=1 to n do
$param_i$.val ← lookup(method_name.env, method_name.name)
make_table($param_i$.name, method_name.env)
od
method_name.addr ← current_point

· Execute [**call** <method_name>(<$param_1$>, <$param_2$>, ..., <$param_n$>) where n ≥1] →
for i=1 to n do
$param_i$.env ← method_name.env
$param_i$.val ← Eval_out[<$param_i$>]
od
return_save(method_name.addr)
control_transfer(method_name.addr)

· Execute [**class** <class_id>]→
class_id.env ← class_id.name
make_table(class_id.name, class_id.env)
class_id.addr ← current_point

· Execute [**public class** <class_id>]→
class_id.env ← **public**
make_table(class_id.name, class_id.env)
class_id.addr ← current_point

· Execute [**class** <class_id><derived_class>]→
class_id.env ← class_id.name
class_id.penv ← derived_id.name
make_table(class_id.name, class_id.env)
class_id.addr ← current_point

· Execute [**public class** <class_id><derived_class>] →
class_id.env ← **public**
class_id.penv ← derived_class.name
make_table(class_id.name, class_id.env)
class_id.addr ← current_point

· Execute [<class_id><object_id>]→
object_id.env ← class_id.name
make_table(object_id, object.env)

· Execute [<modifier><identifier>]→
identifier.name ← identifier
environment.name ←
determine_environment(identifier.name)
identifier.env ← environment.name
make_table(identifier.name, identifier.env)

· Execute [<modifier><$identifier_1$>, <$identifier_2$>, <$identifier_n$> where n≥1]→
for i=1 to n do
$identifier_i$.name ← $identifier_i$
environment.name ←
determine_environment($identifier_i$.name)
$identifier_i$.env ← environment.name
make_table($identifier_i$.name, $identifier_i$.env)
od

· Execute[<object_id> = **new**<class_id>(<$param_1$>,<$param_2$>,...,<$param_n$>) where n≥)]→
object_id.env ← class_id.name
make_table(object_id.name, object_id.env)
member_identifier ←
search_member(class_id.name)
for each member_identifier do
make_table(member_identifier.name, object_id.name)
do
object_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><object_id> = **new** <class_id>(<$param_1$>, <$param_2$>, ..., <$param_n$>)

```
where n ≥1)]→
    object_id.env ← class_id.name
    make_table(object_id.name, object_id.env)
    member_identifier ←
    search_member(class_id.name)
    for each member_identifier do
        make_table(member_identifier.name,
        object_id.name)
    do
    object_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)
```

5. 의미 표현법의 비교

기존의 의미 표현법과 작용식을 비교하기 위해 모든 구문을 표현하는 대신 while 구문에 대한 기존 세 연구의 표현법을 나타내면 다음과 같다.

● DS(Denotational Semantics) 표현법

```
exec  while ( Expr ) Stmt   env scont sto =
  scont1 (env[&break ← scont], sto) where rec
    scont1 = λ(env1, sto1).eval  Expr  env1
    econt sto where
      econt = λ(val, typ, sto2).
        if val = true
        then exec  Stmt  env1 scont1 sto2
        else scont(env, sto2)
```

● ASM(Abstract State Machine) 표현법

```
let stm = (while (exp stm1 ) in
      fst(stm) = fst(exp)
      n  xt(exp) = stm
      n  xt(stm1) = fst(exp)
if task is (while (exp ) stm1 ) then
  if val(exp) = true then
    task := fst(stm1)
  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
```

의미 표현법을 비교하는 방법은 여러 가지가 있는데 기존에 비교한 연구[2, 3, 5]들을 중심으로 판독성(Readability), 지능성(Intelligibility), 모듈성(Modularity), 확장성(Extensibility), 응용성(Applicability) 영역에서 기존 연구의 대표적인 세 표기법인 DS, ASM, AS와 제시된 작용식(AE)을 비교하였는데 기존의 세 가지 표기법에 대한 판정은 의미 표현법에 대해 비교하고 연구한 논문[2, 4]을 바탕으로 정리하였다.

(표 1) 의미 표현법 비교

표현법	판독성	지능성	모듈성	확장성	응용성
DS	낮음	낮음	낮음	낮음	보통
ASM	보통	보통	보통	높음	높음
AS	보통	높음	높음	높음	낮음
AE	높음	높음	높음	높음	높음

본 연구에서 제시한 작용식은 다섯 가지 영역에서 모두 좋은 평가를 받고 있다. 제시된 작용식은 연산 의미론(operational semantics)을 근거로 속성문법을 확장하여 변형한 것으로 변수 값이나 환경 등의 8가지 속성을 사용하여 간단하고 쉽게 동적 의미를 표시하고 있고 8가지 모듈을 사용하여 모듈성을 높이고 있다. 언어를 확장했을 때 추

가의 모듈을 정의하여 동적 의미를 명세할 수 있다. OBLA 언어뿐 만아니라 다른 객체 지향 언어의 의미를 표현할 경우 작용식의 구문만 변경하면 되므로 다른 객체 지향 언어의 의미를 쉽게 명세할 수 있다.

작용식을 사용하여 명령형 언어에 대해 해석기를 구현한 논문[1]에서 알 수 있듯이 본 작용식은 번역기를 쉽게 만들 수 있도록 보다 실제적이고, 구체적이고, 명확한 의미 구조로 명세된 것을 알 수 있다.

6. 결론

본 논문에서는 OBLA라는 새로운 객체 지향 언어를 정의하고 연산 의미론에 근거하여 이 언어에 대한 정적이고 동적인 의미를 명세하는 새로운 작용식을 정의하였다. 작용식은 OBLA 언어뿐만 아니라 일반적인 객체 지향 언어를 위한 정적이고 동적인 의미 구조에 적용할 수 있도록 명세하고 있다.

본 논문의 작용식은 여러 가지 장점이 있다. 제시된 작용식은 연산 의미론에 근거하여 의미를 정의하여 기존의 표현법과 비교할 때 아주 간단하고 명확하다. 작용식의 구문영역에 다른 소스 언어를 사용하면 그 언어에 대한 정적이고 동적인 의미를 얻을 수 있다. 작용식에 표현된 의미 명세에 따라 언어를 구현하면 그 언어에 대한 번역기를 쉽게 만들 수 있다. 소스 언어가 확장될 경우 작용식을 확장할 수 있고 확장된 구문에 대한 의미를 표현할 수 있다.

판독성, 지능성, 모듈성, 확장성, 응용성 영역에서 기존 연구의 세 표기법인 DS, ASM, AS와 제시된 작용식(AE)을 비교했을 때 제시된 작용식이 다섯 영역 모두에서 좋게 평가되고 있어 제시된 작용식이 기존의 표현법보다 우월함을 확인할 수 있다.

참 고 문 헌

[1] 한정란 "작용 식 기반 점진 해석" Ph. D Thesis 이화여대 1999.

[2] David A. Watt and Deryck F. Brown, "Formalising the Dynamic Semantics of Java"

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

[4] Daniel Wasserrab , Tobias Nipkow, Gregor Snelting and Frank Tip, "An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++, OOPSLA'06 October 22-26, 2006.

[5] J. Alves-Foss, editor. Formal Syntax and Semantics of Java, Vol 1523 of Lecture Notes in Computer Science. Springer-Verlag, 1999.

[6] Glynn Winskel, The Formal Semantics of Programming Languages.

[7] Concepts of programming languages, Robert W. Sebesta, Addison Wesley

[8] Frank G. Pagan, Formal Specification of Programming Language, Prentice-Hall, Inc.

◐ 저 자 소 개 ◑

한 정 란(Jung Lan Han)
1985년 이화여자대학교 전자계산학과 졸업(학사)
1987년 이화여자대학교 대학원 컴퓨터학과 졸업(석사)
1999년 이화여자대학교 대학원 컴퓨터공학과 졸업(박사)
1999~현재 협성대학교 경영학부 경영정보 교수
관심분야 : 전자상거래, e-CRM, XML, 웹 서비스, 웹 2.0
E-mail : jlhan@uhs.ac.kr