함수를 포함한 IL 언어의 실행적 의미구조

신 승 철[†] · 노 상 훈[††]

요 약

PLC와 같은 특수 목적 제어기나 모션 제어기 등을 프로그램하기 위해 제공되는 제어 언어의 표준은 IEC61131-3이다. 이 표준 언어의 하나인 IL(Instruction List)은 어셈블리 수준의 언어이지만 고수준 언어의 특징들도 가지고 있다. 본 논문에서는 IL의 정형적인 의미구조를 실행적 의미구조를 이용하여 정의한다. 기존의 IL 의미구조들은 함수와 함수블록을 포함하지 않는 기본 프로그램만을 대상으로 하기 때문에 실용적이지 못하다. 우리는 함수와 함수블록을 포함하는 IL 의미구조를 정의하였다.

키워드 : IEC 61131-3, 인스트럭션 리스트, 의미구조, 시뮬레이터

Operational Semantics for Instruction List with Functions

Seungcheol Shin[†] · Sanghoon Rho[††]

ABSTRACT

IEC61131-3 is the standard of control languages in which special purpose controllers and motion controllers such as PLC can be programmed. IL(Instruction List), one of the standard languages, is in assembly level but has some high-level features. This paper describes a formal semantics of IL in operational style. Previous works on IL semantics do not include functions and function blocks, which is not so practical. We define IL semantics including functions and function blocks.

Key Words : IEC 61131-3, Instruction List, Semantics, Simulator

1. 서 론

오래전부터 최근에 이르기까지 산업자동화, 지능형 빌딩, 임베디드 시스템, 유비쿼터스 환경 등 다양한 제어 시스템에 PLC(Programmable Logic Controller)라고 불리는 제어기(프로세서)를 사용해왔다[13, 14]. PLC는 기존의 물리적 회로에 의해 작동하도록(hardwired logic) 만들어지던 제어장치를 메모리상의 프로그램에 의해 작동할 수 있게(softwired logic) 대체하는 장치이다. 때문에 기존의 방식에서 사용하던 회로도는 PLC에서 프로그램으로 대체된다. 이 제어기를 위한 프로그래밍 언어의 표준을 정의한 것이 IEC61131-3이다 [2].

IEC61131-3은 제어 프로그래밍을 위한 다섯 가지 언어 LD(Ladder Diagram), ST(Structural Text), IL(Instruction List), FBD(Function Block Diagram), SFC(Sequential Function Chart)를 제시한다 [2]. 본 논문은 그 중에서 IL 언어에 대한 정형적인 의미구조(formal semantics)를 정의한다. 이 언어는 PLC 시스템의 기본 구성 단위인 프로그램, 함수블럭, 함수와 같은 POU(Program Oraganization Unit)의 본체를 작성하는데 사용하는 언어이다. IL의 의미구조는 표준 문서에서도 정의 하고는 있지만 [2], 여기에서는 모호성을 지닌 자연어로 정의되어 있고 충분히 자세히 설명하고 있지 않다. 지금까지 만들어진 IL 프로그램 시뮬레이터(또는 가상기계)는 견고하고 정형적인 의미구조를 바탕으로 만들어지지 않았기 때문에 동작의 오류가 존재할 잠재성이 있다. 또한 정형적인 의미구조를 제시하는 연구의 경우에도 함수와 함수블록 등을 포함하지 않는 기본 언어에 대해서만 다루고 있다. 본 논문은 실용적인 프로그램에 필수적인 함수와 함수블록을 포함하는 IL에 대하여 정형적인 의미구조를 정의한다. POU의 선언은 IL의 범위를 벗어나므로 본 논문에서 다루지 않는다. POU의 인터페이스 정의는 통상적인 프로그래밍 언어와 같이 문자적으로 작성하기도 하지만, 많은 경우에는 대화식 GUI를 이용한다.

IL 언어는 어셈블리어와 유사한 저수준의 언어이다. 하지만 IL은 특정 PLC나 프로세서에 종속하지 않기 때문에 제어언어 처리에서 중간 언어로서 유용하며, IL 자체로써도 훌륭한 프로그래밍 언어이다. 제어 프로그래머들이 IL 프로그램을 개발할 때에 컴파일한 목적 코드를 직접 특정 PLC나 프로세서에 업로드하면서 디버깅하는 것은 매우 비효율적이다. 그래서 IL 프로그램을 컴퓨터에서 시뮬레이션 할 수 있는 IL 가

※ 이 논문은 2006년도 한국기술교육대학교 신임교수 연구비 지원에 의하여 연구되었음.
† 정 회 원 : 한국기술교육대학교 인터넷미디어공학부 조교수
†† 정 회 원 : 한국기술교육대학교 정보미디어공학과 석사과정
논문접수 : 2007년 7월 31일, 심사완료 : 2007년 11월 12일

상기계를 많이 사용한다 [12].

IL 가상기계를 정형적으로 설계하여 구현하면 가상기계의 올바른 동작을 보장할 수 있을 뿐 아니라 IL 프로그램의 특성을 분석하고 증명하는 메커니즘을 개발하는 데에도 사용할 수 있다. 본 논문은 IL 가상기계를 정형적으로 정의하기 위하여 IL 언어의 실행적 의미구조(operational semantics)를 제시한다.

이를 위해서 논문의 나머지 부분은 다음과 같이 구성된다. 2절에서는 관련 연구를 설명하고, 3절부터 5절까지는 IL 언어의 구문구조와 의미구조를 차례로 정형적으로 정의한다. 6절에서는 간단한 예제를 보여주며, 마지막 7절은 결론이다.

2. 관련 연구

IL 언어의 의미구조를 제시하는 작업은 많이 알려져 있지 않다. IEC61131-3 표준문서는 정형적인 방법이 아닌 자연어로 간단하게 의미들을 정의한다 [2].

Ralf Huuck은 IL 언어에 대해 분석을 목적으로 하여 정형적 방법으로 의미구조를 정의하고 있지만, 분석을 위해 주요한 연산자 정도의 매우 간략화 된 구문에 대해서만 의미구조를 정의하였다 [4,5]. 또한, 타입 역시 오직 정수(int)와 부울(bool) 타입만을 가정한다. PLC는 스캔 사이클(scan cycle)이라고 불리는 주기적이고 반복적인 단위로 IL 프로그램을 실행한다. 하나의 스캔 사이클에서만 효과를 주는 다른 연산자들과 달리, 래치 연산자는 새로운 스캔 사이클에서도 그 효과가 유지되어야 한다. 하지만 Huuck의 정의는 이 효과를 표현하기 위한 특별한 장치를 고려하지 않았다.

본 논문의 IL 언어는 Huuck의 정의를 포함할 뿐 아니라, 함수와 함수블록에 관한 정의를 추가적으로 제공한다. 또한 유도된 자료형(derived data type)과 위에서 지적한 래치 연산자에 대한 특성 역시 고려한다.

함수 호출형 언어에는 SECD 기계를 사용할 수 있지만, 이 기계는 기본적으로 동적 메모리 관리를 이용하는 환경을 고려하여 스택을 이용한 호출구조를 이용하고 있다. 반면, IEC 61131-3은 기본적으로 스택 등 동적 메모리를 배제하고 정적 메모리 관리를 사용하기 때문에, SECD는 IL 언어의 의미구조에는 적합하지 않다 [6].

IL 언어는 바이트코드에 가까운 것으로, 이와 관련하여 Peter Bertelsen의 자바 바이트코드의 의미구조를 정의하는 논문이 이런 정의에 대하여 잘 보여준다 [10]. 그러나 JVM은 스택 기반 기계인 반면 PLC는 누산기(accumulator) 기반 기계라는 차이점이 있다.

본 논문에서는 핵심적인 부분만을 정의하며 중복적이거나 중요하지 않은 부분은 생략한다. 더 자세한 정의는 [7]에서 찾을 수 있다.

3. IL 언어의 구문구조

본 논문의 IL 언어는 표준문서에 정의된 IL이 갖고 있는 모든 구문들을 포함하지는 않는다. 일부 전처리로 손쉽게 다른 구문으로 변환될 수 있는 문법들을 제외하여 간략화 했다. 또한, 추상구문구조로 정의함으로써 의미구조가 지나치게 복잡하게 되는 것을 피하였다. 간략화되지 않은 원래의 IL 언어의 정의는 IEC 61131-3 표준문서에서 찾을 수 있다 [2].

IL의 구문구조:

ins_list	::=	*ins* | *ins ins_list*
ins	::=	*simple_op operand* | *fun_name* {*operand*} | *jump_op label* | *call_op x* {*id* := *operand*} | *return_op*
operand	::=	*lit* | *x*
simple_op	::=	**LD** | **LDN** | **ST** | **STN** | **NOT** | **S** | **R** | **AND** | **OR** | **XOR** | **ANDN** | **ORN** | **XORN** | **ADD** | **SUB** | **MUL** | **DIV** | **MOD** | **GT** | **GE** | **EQ** | **LT** | **LE** | **NE**
jump_op	::=	**JMP** | **JMPC** | **JMPCN**
call_op	::=	**CAL** | **CALC** | **CALCN**
return_op	::=	**RET** | **RETC** | **RETCN**
x	::=	*id* | *x.id* | *x*[{*index*}]
index	::=	*lit* | *x*

식별자를 가정하는 집합:
- ID = 모든 식별자의 집합
- FUN_ID = 선언된 함수 식별자의 집합

리터럴과 레이블을 가정하는 집합:
- LIT = 모든 리터럴의 집합
- LAB = 모든 레이블의 집합

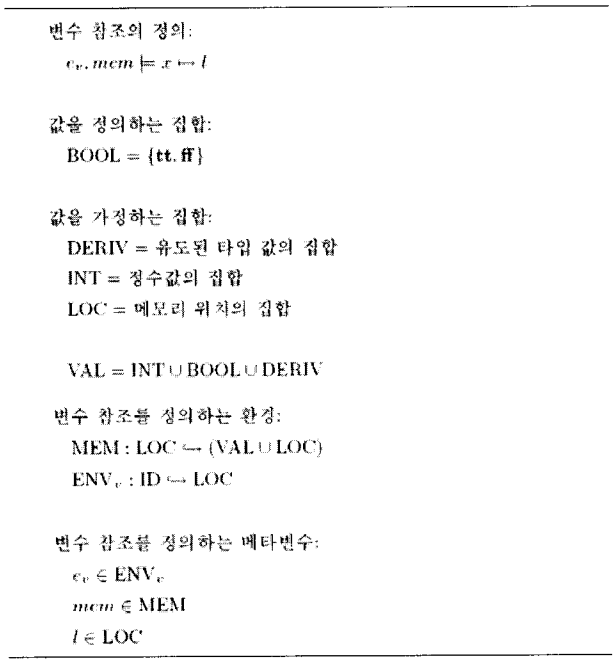
IL 구문구조를 정의하는 메타변수:
- *lit* ∈ LIT
- *label* ∈ LAB
- *id* ∈ ID
- *fun_name* ∈ FUN_ID

(그림 1) IL 언어의 구문구조

본 논문의 간략화된 IL 언어의 구문구조는 (그림 1)에 나온다. IL 프로그램은 일반적으로 "*연산자 피연산자*" 로 된 명령어의 리스트로 표현된다. IL 명령어는 연산자의 종류에 따라서 피연산자를 하나만 포함하거나 피연산자를 가지지 않는다. 예외적인 경우는 함수/함수블록의 호출문이다. 함수 (*fun_name* ...)와 함수블록(*call_op* ...)은 1개 이상의 피연산자들을 취한다.

본 논문은 IL 구문구조의 간략화에 해당하는 전처리 단계를 가정한다. 이 단계에서는 레이블 선언과 괄호 표현식이 제거된다. 간략화 후에 프로그램 상의 레이블은 분기문의 피연산자로만 나타나며, 제거된 레이블 선언들은 레이블 문맥(context)을 구성한다. 또한, 괄호표현식의 구현에는 스택이 필요한데, 이는 임시변수를 도입하여 제거한다고 가정한다.

PLC는 정적 메모리 관리환경을 사용하고 스택을 배제하기 때문에, 괄호표현식 뿐만 아니라 함수/함수블록 호출에 따른 복귀 주소(return address)도 각 POU의 환경에 저장하는 방법을 사용한다



(그림 2) 변수 참조 관계

또한 IEC 61131-3에서는 재귀호출이 허용되지 않으며, 함수블록이 독립적이고 정적인 인스턴스(instance)를 갖는다. 때문에, 효율성을 위해 도입되는 명시적인 스택 변수 선언이나 앞에서 언급한 구문들을 제거하면 스택을 배제한 정의할 수 있다.

4. IL 언어의 의미구조를 정의하는 집합들

IEC 61131-3은 다양한 자료형(data type)들을 제공하는데, 이들간의 타입, 오버로딩, 참조 등은 의미구조를 복잡하게 한다. 본 논문에서는 함수 호출에 관한 부분에 집중하기 위해 자료형을 정수와 부울만을 가정한다. 하지만 구조적 자료형(structured data type)이나 배열 자료형(array data type)과 같은 유도된 자료형은 타입, 참조, 오버로딩 문제와는 독립적으로 다루어질 수 있으므로 포함한다. 특히 구조적 자료형은 함수블록의 인터페이스 변수(interface variable)의 접근을 위한 한 형태로 사용되기 때문에 반드시 포함해야 한다.

변수로부터 대응되는 주소를 찾는 일은 매우 빈번한데, 배열 변수나 구조적 변수와 같은 유도된 변수들은 이를 복잡하게 만든다. 이 복잡성을 줄이기 위해, 변수로부터 주소(location)를 참조하는 것에 관한 정의를 (그림 2)와 같이 별도로 정의한다.

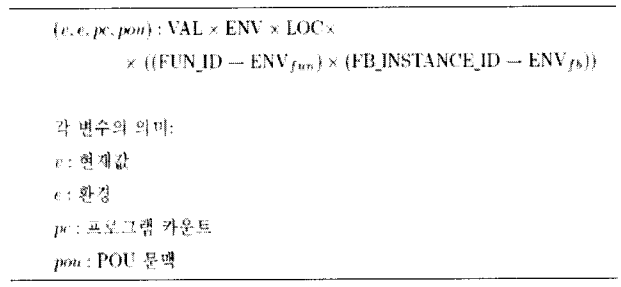
변수 참조는 주어진 변수 환경(e_v)과 메모리(mem)로부터 변수 x에 대응하는 메모리 주소(location) l을 찾아낸다. 여기서 변수 환경은 변수 명칭을 주소로 대응하는 함수이고, 뒤 쪽에서 더 자세하게 설명하게 될 종합환경의 요소이기도 하다. 메모리는 각 주소에 대해 값이 대응되는 상태이다.
변수 x는 (그림 1)의 구문구조에 정의 된 것과 같이 단순한 식별자로만 되어 있을 수도 있지만, 앞에서 언급하였듯이 점(.)에 의해 구분되는 구조적 변수이거나 인덱스를 갖는 배열 변수일 수도 있다. 전자의 경우라면 변수 환경만으로 변수 참조를 할 수있다. 하지만 배열과 구조적 변수를 하나의 값으로 가정하고 있기 때문에 후자의 경우라면 메모리 상태도 필요로 하게 된다. 때문에 변수 참조는 변수 환경과 메모리 둘 모두가 필요하다.

이런 방식으로 변수 참조로 분리 시켜두는 것은 변수로부터 주소가 올바르게 구해진다는 가정 하에서, 명령어의 실행에 대한 정의를 훨씬 간결하게 할 수 있게 한다. 변수의 주소 계산에 대한 정의는 여기에서는 생략한다. 하지만 이에 대한 자세한 정의는 [7]에서 찾아 볼 수 있다.

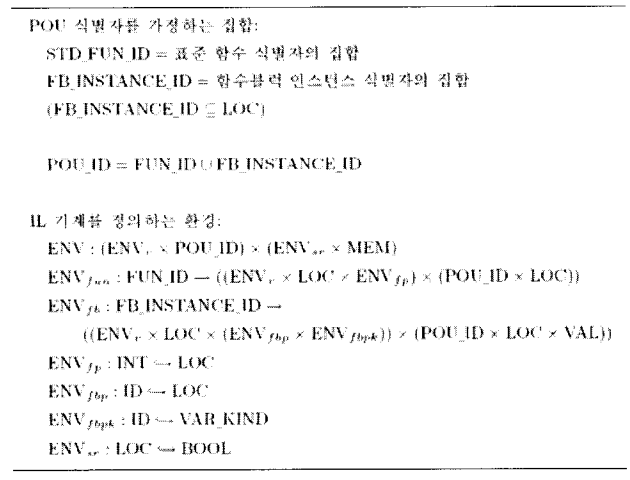
IL 기계의 구성(configuration)은 (그림 3)과 같이 4가지 요소로 이루어진다. 첫 번째 요소인 누산기는 표준문서에서 현재값(current result)으로 설명하고 있다. 이는 이전 연산에 의한 결과 값을 담고 있으며, 다음 연산을 하기 위한 입력 값으로 사용된다. 앞에서 구문구조의 간략화 과정에서 괄호표현식이 제외되었다고 언급했는데, 이를 효율적으로 추가하기 원한다면 누산기를 스택의 형태로 확장하여야 한다.

두 번째 요소인 환경 e는 프로그램을 실행하는데 필요한 여러 가지 문맥과 상태가 들어 있다. (그림 4)는 이에 대해 상세한 정의를 담고 있다.

종합환경 ENV는 여러 가지 부분환경의 묶음으로 이루어진다. 이것은 크게 (ENV_v × POU_ID)와 (ENV_{sr} × MEM)의 두 부분으로 구성된다.



(그림 3) IL 기계의 구성



(그림 4) IL 기계의 구성을 정의하는 집합

전자는 지역환경으로써 호출에 의해서 변경되는 환경으로 함수를 호출하거나 호출된 함수가 종료될 때 변경되는 환경이다. 함수를 호출하면 변수들에 대해서 새로운 문맥을 갖는 것과, 지금 실행하고 있는 POU가 달라지는 것은 명백하다. 복귀 주소는 정적으로 존재하는 각 POU에 기록된다. 이를 통해 현재 POU가 무엇인지 앎으로써 호출이 완료되고 복귀할 때 어디에서 이 정보를 가져올지를 알 수 있다. 이 정보는 스택 기반 언어에서 활성화 레코드를 가리키는 스택 포인터와 유사한 역할을 한다.

종합환경의 두 번째 부분은 전역환경으로써 호출이나 호출의 종료에 영향을 받지 않는다. 대신, 이 환경들은 저장이나 래치 명령어의 영향을 받는다. ENV_{sr} 은 단일의 IL 프로그램 내로만 국한한다면 별로 의미가 없는 환경이다. 하지만, IL 프로그램이 다시 실행 될 때 메모리에 기본적으로 설정되어야 하는 값이 저장 된다. 이 환경은 더 상위에서 본다면 IL 프로그램에 대한 실행 결과라고 볼 수도 있다.

IL 기계의 세 번째 요소인 프로그램 카운트 *pc*는 현재 실행하고 있는 명령어의 주소를 가리킨다. 명령어를 기계에 직접 표현하는 다른 정의와는 달리 [4,5], 이런 간접적인 표현 방법을 사용한 것은 분기문이나 호출문과 같은 제어 흐름에 관련된 구문의 의미를 명시적으로 반영하기 위해서이다. 이해를 쉽게 하기 위해 프로그램 카운트에 대한 기본 산술 단위는 하나의 명령어 길이라고 가정한다. 즉, *pc*+1 은 *pc* 의 다음 명령어 위치를 의미한다.

마지막 요소인 POU 문맥 *pou*는 POU 호출과 긴밀하게 관련되는 문맥이다. IEC 61131-3에는 앞에서도 언급한 POU(Program Organization Unit)라는 유닛이 정의되는데, 여기에는 프로그램, 함수블록, 함수의 세 유형이 있다. 프로그램은 하드웨어 의존적인 유닛으로 본 논문에서 어느 정도 고려는 하지만 자세한 부분은 논외로 한다.

함수블록과 함수는 다소 차이가 있다. 함수는 완전히 부수효과(side-effect)를 배제하는 반면, 함수블록은 부수효과를 허용하는 것으로 함수와는 달리 인스턴스를 이용하여 호출되게 된다. 이 인스턴스는 변수를 선언하는 방법과 동일한 방법으로 선언한다. 또한, 함수블록의 인터페이스 변수들은 마치 구조체처럼 취급되어 함수블록 인스턴스로부터 점(.) 한정자를 통해 접근도 가능하며, 호출시 피연산자들이 매개변수 할당식의 형태로 전달된다. 이는 매개변수 순서에 의해 피연산자가 구분되는 함수 호출과 구별된다. 때문에 함수블록은 함수와 다르게 취급되어야 한다. 이런 목적으로 *pou*는 함수와 함수블록에게 각각 적절한 다른 종류의 호출 환경을 제공해준다. 이에 대한 자세한 설명은 뒤 쪽의 함수/함수블록의 호출과 복귀에 대한 정의에서 한다. IL 기계는 기본적으로 여러개의 POU가 얽혀서 돌아가기 때문에, 환경과 POU 문맥과 같은 것들은 로더를 통해서 미리 구성되어 있어야 하며, 이는 전처리 단계에서 구성된다고 가정한다.

IL의 의미구조를 정의하기에 앞서 몇 가지 함수들을 먼저 정의한다. 먼저 정의하는 두 함수는 표준함수 호출을 처리해주는 *std_fun_routine* 함수와 리터럴을 값으로 바꾸어주는 *lit_val* 함수이다. 표준함수는 타입변환과 같은 사용자 정의로는 정의될 수 없는 기능들을 포함하고 있기 때문에 별도로 처리할 수 있도록 하였다. 하지만, 본 논문의 초점을 벗어나므로 가정하고 자세한 정의는 생략한다. 본 논문에서는 제한적인 타입만을 허용하기 때문에 리터럴과 값 역시 관심대상은 아니다. 하지만, 이 함수들을 확장함으로써 쉽게 본 논문의 IL 언어를 확장할 수 있다.

instr 함수는 주어지는 프로그램 카운트에 대해 해당 위치에 있는 명령어를 가져오는 함수이다. 명령어란 연산자와 피연산자를 모두 포함하는 하나의 완전한 문장을 의미한다.

to_value 함수는 변수나 리터럴을 값으로 바꾸어 주는 함수로, 변수 혹은 리터럴 그 어느 쪽도 될 수 있는 피연산자를 통합적을 다루기 위해 정의한 함수이다. 이들을 개별적으로 다루면, 언어의 정의가 매우 복잡해지며 특히 함수 호출과 같이 변수와 리터럴을 혼용할 수 있는 구문에 대해서는 정의하기가 매우 어려워진다.

*to_deep_loc*는 변수에 대해 주소 안의 값이 주소라면 그 -또한 값이기도 한-주소를 넘겨주고, 이미 값이라면 현재 변수의 주소를 넘겨주는 함수이다. 즉, 주소값이 아닌 값을 가리키는 주소를 알려주는 함수이다.

변수의 선언 형식에 따라 VAR_KIND 집합으로 정의된 것은 변수의 입/출력 제한 형식을 의미하는데, 이 중 **VAR_IN_OUT**으로 선언된 변수는 참조(reference) 변수로써 다른 변수와는 구별된다. 함수블록에 **VAR_IN_OUT**로 선언된 매개변수에 넘겨주는 변수 역시 **VAR_IN_OUT**으로 선언된 경우라면, 넘겨주는 변수의 주소는 중요하지 않다. 정말 중요한 것은 그 안에 들어 있는 -더 상위 POU에서 전해진 변수의- 주소이다. 이 경우 만약 변수 내의 값이 아닌 변수 자체의 주소를 할당하게 된다면, 매번 실제 값에 접근하기 위해 임의의 깊이의 탐색을 필요로 하게 된다. 즉, 매번 이 변수에 접근을 하려고 할 때마다 불필요한 연산을 더 요구하게 된다. 또한 참조 변수간에 상호 참조적인 할당은 해당 변수의 접근 그 자체로 무한 루프의 위험을 갖는다. 때문에 이러한 참조 변수의 할당에 대한 제약은 필수적이다.

5. IL 언어의 의미구조

앞의 정의를 바탕으로 IL 언어의 의미구조를 비교적 간략하게 정의할 수 있다. (그림 6)은 저장연산자(**ST**)와 적재연산자(**LD**)의 의미구조를 보여준다. 저장연산자는 피연산자로 주어지는 변수에 누산기의 값을 저장한다. 이 연산자는 오직 메모리와 프로그램 카운트 *pc* 만을 변화 시킨다. 적재연산자는 저장연산자와 반대로 피연산자를 누산기에 저장한다. 적재연산은 저장연산 보다 더 간단한 연산으로 환경에 부수효과를 일으키지 않는다.

이 두 연산자는 다른 연산자에서도 중복적으로 나타나는 부분을 가지고 있다. 이들을 살펴봄으로써 본 논문에서 어떤 방식으로 IL 언어의 의미구조를 정의하고 있는지 알 수 있다.

저장연산자는 하나의 변수를 피연산자로 받고 있다. 변수

에 값을 저장한다는 것은 엄밀히는 메모리상에서 해당 변수가 가리키는 주소에 값을 저장함을 의미한다. 때문에 먼저 변수 x의 주소 l을 알아내야 한다. 이는 앞 절에서 언급하였던 변수 참조를 통해서 얻을 수 있다. 이렇게 메모리에서 얻은 주소에 누산기의 값 v를 새롭게 저장한다.

적재 연산은 반대로 피연산자로부터 값을 얻어야 한다. 리터럴의 경우는 앞서 정의 했던 *lit_val* 함수로 얻을 수 있고, 변수의 경우는 저장연산자의 경우와 마찬가지로 주소를 먼저 알아낸 뒤, 이를 이용해 다시 메모리로부터 알아내야 한다. 이는 단순한 변수의 경우이고 만약 변수가 **VAR_IN_OUT** 으로 선언된 변수라면 현재의 값은 주소이므로, 이를 가지고 다시 메모리를 탐색해야 한다. 이런 총 3가지의 시나리오가 피연산자로부터 값을 얻어내는 동작에 존재한다. 많은 명령어에서 사용되는 이런 형태의 피연산자의 참조는 의미구조 정의를 매우 복잡하게 만든다. 때문에 본 논문에서는 *to_value* 라는 함수를 이용해 통합하고 있다.

IL 언어의 연산자는 **LDN**, **STN**과 같이 N으로 끝나는 연산자가 상당수 존재하는데, 이는 피연산자에 부정(negation)을 취한뒤 연산하는 연산자이다. 이들에 대한 정의는 크게 차이가 없으므로 생략한다.

(그림 7)은 산술연산자의 대표적 정의로 제시한 **ADD** 연산자의 정의는 다른 산술연산자(**SUB, MUL, DIV, MOD**)와 비교연산자(**GE, GT, EQ, NE, LE, LT**) 그리고 단항을 취하는 논리연산자(**AND, OR, XOR**)에서도 거의 동일하게 적용된다. 이 연산의 대부분은 앞서 정의한 저장연산자와 동일하다. 단지, 피연산자의 값을 그대로 누산기로로 올리는 것이 아니라 현재의 누산기 값과 피연산자로 연산한 결과를 올린다는 점의 차이이다. 이 연산의 경우는 더하기 연산을 하고 있다.

다른 연산자들은 그 연산자의 이름으로부터 쉽게 그 의미를 추론할 수 있고, 대표로 제시된 [ADD Operation] 과 같은 방식으로 정의할 수 있으므로 생략한다.

앞의 단항 연산자와 달리 **NOT**은 피연산자가 없는 논리연산자이다. 이 연산자는 원래 비트열 타입의 값에 대해 비트 단위의 부정(bitwise negation)을 취하는 연산자이지만, 본 논문에서는 논리연산을 위해 참(tt)/거짓(ff) 만을 표현하는 1비트의 부울 타입만을 가정하므로 토글의 역할로 정의된다. (그림 8)에 정의된 것과 같이 이 연산자는 단지 누산기에 간단히 부정연산을 취하는 동작을 한다.

IL 의미구조를 정의하는 메타변수:

$vl \in \mathrm{VAL} \cup \mathrm{LOC}$

$callee\_id \in \mathrm{POU\_ID}$

$caller\_id \in \mathrm{POU\_ID}$

IL의 의미구조를 정의하는 집합:

OPERAND = *operand* 구문구조로 정의되는 집합

INS = *ins* 구문구조로 정의되는 집합

VAR_KIND = {**VAR_INPUT**, **VAR_OUTPUT**, **VAR_IN_OUT**, **VAR_NORMAL**}

IL의 의미구조를 정의하는 가정된 집합:

$\mathrm{VAL\_LIST} = \{< v_0, \ldots v_n > \mid v_0, \ldots v_n \in \mathrm{VAL}\}$

($< v_0, \ldots, v_n >$ 는 순서를 갖는 값 리스트)

가정되는 레이블 문맥:

lab = label에 대해 대응되는 프로그램 카운트를 갖는 맵핑

IL의 의미구조를 정의하는 가정된 함수:

$std\_fun\_routine : \mathrm{STD\_FUN\_ID} \times \mathrm{VAL\_LIST} \rightharpoonup (\mathrm{VAL} \times \mathrm{ENV})$

$lit\_val : \mathrm{LIT} \rightharpoonup \mathrm{VAL}$

$instr : \mathrm{LOC} \rightharpoonup \mathrm{INS}$

IL의 의미구조를 정의하는 정의된 함수:

$$to\_value(e_v, mem, lit) = lit\_val(lit)$$

$$to\_value(e_v, mem, x) = \begin{cases} v \text{ if } e_v, mem \models x \mapsto l \\ \quad \wedge vl = mem(l) \\ \quad \wedge v = \begin{cases} mem(vl) \text{ if } vl \in \mathrm{LOC} \\ vl \text{ if } vl \in \mathrm{VAL} \end{cases} \end{cases}$$

$$to\_deep\_loc(e_v, mem, x) = \begin{cases} l' \text{ if } e_v, mem \models x \mapsto l \\ \quad \wedge vl = mem(l) \\ \quad \wedge l' = \begin{cases} l \text{ if } vl \in \mathrm{VAL} \\ vl \text{ if } vl \in \mathrm{LOC} \end{cases} \end{cases}$$

(그림 5) IL 의미구조를 정의하는 집합

[ST Operation]

$$(v,\ e,\ pc,\ pou) \longrightarrow (v,\ e',\ pc+1,\ pou)$$

$$\text{where} \begin{cases} instr(pc) = \mathbf{ST}\ x \\ e = ((e_v, this\_id), (e_{sr}, mem)) \\ e_v, mem \models x \mapsto l \\ e' = ((e_v, this\_id), (e_{sr}, mem[l \mapsto v])) \end{cases}$$

[LD Operation]

$$(v,\ e,\ pc,\ pou) \longrightarrow (v',\ e,\ pc+1,\ pou)$$

$$\text{where} \begin{cases} instr(pc) = \mathbf{LD}\ operand \\ e = ((e_v, this\_id), (e_{sr}, mem)) \\ v' = to\_value(e_v, mem, operand) \end{cases}$$

(그림 6) ST와 LD 연산자의 의미구조

[ADD Operation]

$$(v_0,\ e,\ pc,\ pou) \longrightarrow (v_0 + v_1,\ e,\ pc+1,\ pou)$$

$$\text{where} \begin{cases} instr(pc) = \mathbf{ADD}\ operand \\ e = ((e_v, this\_id), (e_{sr}, mem)) \\ v_1 = to\_value(e_v, mem, operand) \end{cases}$$

(그림 7) 단항 산술연산자의 의미구조

[NOT Operation]

$$(v,\ e,\ pc,\ pou) \longrightarrow (\neg v,\ e,\ pc+1,\ pou)$$

where $instr(pc) = \mathbf{NOT}$

(그림 8) 무항 논리연산자의 의미구조

앞에서 언급한 바와 같이 래치 연산은 IL 기계 보다 더 상위에 있는 기계의 동작과 연관이 있는 연산이다. PLC에는 POU로써의 프로그램에 태스크(task)를 할당하여 어떠한 일련의 동작을 수행하는데, 작업과 연관된 프로그램은 주기적으로 혹은 특정 조건에 의해서 여러 번 호출 될 수 있다. 이 때 현재의 IL 기계의 수행에서 뿐만 아니라 다음의 호출 수행에까지 이어지게 되는 설정 값을 주는 것이 바로 래치 연산자이다.

다른 연산과 차이가 나는 단적이 예를 들어 보자면, **S** *x*로 **tt**로 설정된 *x*는 뒤에 **ST** *x*에 의해 다른 어떤 값으로 설정되더라도, 다음 스캔 사이클에서는 다시 **tt**로 돌아오게 된다. 이를 다음 스캔 사이클에서 **ff**로 만들려면 오직 **R** 이라는 또 다른 래치 연산을 사용해야 한다.

이렇게 래치 연산의 결과는 현재의 메모리에 설정된 값과는 별개로 프로그램의 수행이 종료되었을 때까지 유지해야 하므로 이에 대한 상태를 저장할 환경이 필요하다. 이것이 앞에서 언급했던 래치 환경 e_{sr} 이다. 연산결과가 현재의 프로그램 실행에서도 적용되도록 메모리를 수정 하는 것과 더불어 래치 환경에도 동일하게 적용한다. 태스크나 리소스등과 같이 IL 언어 보다 더 상위의 기계를 정의하려고 한다면, 이 정보를 활용하면 된다.

래치 연산자의 누산기가 **tt**로 되어 있는 것은, 이 연산이 누산기의 값이 tt인 경우에만 효과가 있음을 뜻한다. 만약, **ff**라면 단순히 프로그램 카운트만 증가시킬 것이다.

함수는 보통의 다른 연산자와 같이 함수이름이 연산자 위치에 오며, 연산자처럼 사용된다. 즉, 함수이름 뒤에 오는 피연산자뿐만 아니라 누신기의 값도 하나의 피연산자로 취한다. 하지만 기본 연산자들과 달리 함수는 여러 개의 피연산자를 갖을 수 있다. $\{operand_i\}_{i=1..n}$ 는 $operand_1, ..., operand_n$ 의 피연산자 리스트를 의미한다. 함수는 이 피연산자들과 누산기의 값 v_0를 포함해 n+1개의 피연산자로 연산을 수행한다.

표준함수의 식별자는 (그림 5)에서 가정하였듯이 STD_FUNS에 속해 있다. 이를 통해 표준함수와 사용자함수가 구별이 되며, STD_FUNS에 속해 있는 함수의 호출은 앞에서 언급한 *std_fun_routine*를 통해 빌트인(built-in)으로 만들어지는 함수를 통해 연산된다. *std_fun_routine* 함수는 *callee_id*를 이름으로 갖는 함수 f 대해 $f(< v_0, ..., v_n >)$를 결과로 돌려주는 함수이다. 이 때 함수의 입력 $< v_0, ..., v_n >$는 순서를 갖는 값 리스트이다. 함수의 입력 값은 그 순서가 의미를 갖기 때문이다.

(그림 11)에서 정의하고 있는 사용자함수의 의미구조는 피연산자의 순서의 차이를 보여주고 있다. 선언 부분은 IL 언어의 범위를 벗어나기 때문에 본 논문에서는 환경으로 주어진다고 가정을 하고 있는데, 함수의 인터페이스 변수로 선언된 순서와 피연산자의 순서가 관련이 있다. 누산기의 값은 첫 번째 선언된 변수로 전달되며, 나머지 변수에는 함수이름 뒤에 오는 피연산자들이 순서대로 전달된다. 때문에 호출되는 함수(callee)로 값을 전달하기 위해 순서에 따라 대응 되는 인터페이스 변수의 주소를 먼저 가져와야 한다. 이 정보를 제공해 주는 환경이 e_{fp} 이다. 반면 함수블록은 순서가 아니라 형식 매개변수(formal parameter)의 이름과 관련이 있어 다른 종류의 환경이 필요하다. 때문에 POU 문맥에는 *fun*, *fb* 두 종류의 환경을 가지고 있다. 함수 호출에서는 함수 문맥 *fun*이 *callee_id*로부터 대응되는 호출 환경을 가져온다.

호출 환경 중 e_{fp}를 제외한 나머지 환경들은 함수블록의 경우에도 공통적으로 적용되는 환경으로 지역변수(e_v')와 호출되는 POU의 명령어 위치(pc'), 그리고 복귀(*ret_info*)에 관련된 환경이다. 현재의 IL 기계 상태에서 지역변수나 프로그램 카운트는 이들로 교체됨으로써, 함수 호출이 이루어진다.

이와 더불어 호출이 완료되고 복귀할 때를 위해 고려해 주어야 하는 점이 있다. 호출 환경에서 e_v', pc', e_{fp}는 고정적인 환경이지만, 복귀 환경(*ret_info*)은 그렇지 않다. 호출되는 함수의 명령어의 실행이 완료되면 현재의 명령어 위치로 -

[S Operation tt]

$$(\mathbf{tt},\ e,\ pc,\ pou) \longrightarrow (\mathbf{tt},\ e',\ pc+1,\ pou)$$

$$\text{where}\ \begin{cases} instr(pc) = \mathbf{S}\ x \\ e = ((e_v, this\_id), (e_{sr}, mem)) \\ e_v, mem \models x \mapsto l \\ e' = ((e_v, this\_id), (e_{sr}[l \mapsto \mathbf{tt}], mem[l \mapsto \mathbf{tt}])) \end{cases}$$

(그림 9) 래치 연산자의 의미구조

[STD Fun Call]

$$(v_0,\ e,\ pc,\ pou) \longrightarrow (v',\ e',\ pc+1,\ pou)$$

$$\text{where}\ \begin{cases} instr(pc) = callee\_id\ \{operand_i\}^{i\in 1..n} \\ callee\_id \in \text{STD\_FUNS} \\ e = ((e_v, this\_id), (e_{sr}, mem)) \\ \forall_{i\in 1..n} : v_i = to\_value(e_v, mem, operand_i) \\ (v', e') = std\_fun\_routine(callee\_id, < v_0, ..., v_n >) \end{cases}$$

(그림 10) 표준함수 호출의 의미구조

[Fun Call]

$$(v_0,\ e,\ pc,\ (fun, fb)) \longrightarrow (?,\ e',\ pc',\ (fun', fb))$$

$$\text{where}\ \begin{cases} instr(pc) = callee\_id\ \{operand_i\}^{i\in 1..n} \\ callee\_id \in \text{FUN\_ID} \\ e = ((e_v, caller\_id), (e_{sr}, mem)) \\ ((e_v', pc', e_{fp}), ret\_info) = fun\ callee\_id \\ fun' = fun[caller\_id \mapsto ((e_v', pc', e_{fp}), (caller\_id, pc+1))] \\ \forall_{i\in 1..n} : v_i = to\_value(e_v, mem, operand_i) \\ \forall_{k\in 1..n} : l_k = e_{fp}\ k \\ mem' = mem([l_k \mapsto v_k] | k = 0..n) \\ e' = ((e_v', callee\_id), (e_{sr}, mem')) \end{cases}$$

(그림 11) 사용자함수 호출의 의미구조

엄밀히는 호출 다음 명령어- 위치로 복귀하여야 한다. 때문에, 새로운 호출문맥(*fun'*)에서 이 함수의 호출환경은 (그림 11)과 같이 복귀환경이 바뀐 호출환경이 된다.

복귀 환경에는 *caller_id* 와 *pc*+1 의 두 가지가 주어지고 있는데, 전자는 환경을 복원하기 위함이고 후자는 명령어의 위치를 복원하기 위함이다. *caller_id* 만으로는 명령어의 시작 위치 밖에 얻어내지 못하기 때문에 *pc+1* 을 별도로 저장한다.

함수블록의 호출은 함수 호출과 다소 차이가 있는데, 함수가 함수이름을 이용해 호출이 가능한 것과는 달리 함수블록은 인스턴스를 통해서 호출된다. 즉, 변수 선언의 형태로 인스턴스를 만들고, 인스턴스 변수로 부터 함수블록 인스턴스의 식별자를 알아내어야 한다. 함수가 별도의 연산자 없이 함수이름만으로 기본연산자처럼 사용되는 것과는 달리, 함수블록 인스턴스 변수는 **CAL** 이나 **CALC**, **CALCN** 과 같은 함수블록 호출 연산자의 피연산자처럼 사용된다.

이 때, 함수블록 인스턴스는 호출 할 때 매개변수로 전달될 수도 있기 때문에, 인스턴스 변수 이름을 인스턴스의 식별자로 사용하기에는 무리가 있다. 때문에, 함수블록 인스턴스는 단지 호출에만 사용되는 것이 아니라 그 자체로 구조적 변수처럼도 사용되는데, 이 구조적 변수 역시 변수 환경에 주어진다고 가정하고 있다. 본 논문에서는 이 가상구조체의 주소를 함수블록 인스턴스의 주소를 식별자로 사용한다. 그래서 (그림 12)의 정의에서는 먼저 함수블록 인스턴스 변수 x_{callee}로부터 변수 참조의 형태로 *callee_id*를 가져온다고 정의하고 있다. 함수와 함수블록이 큰 차이를 보이는 다른 한 가지는 피연산자의 전달이다. 함수에서는 피연산자의 순서가 중요했지만, 함수블록에서는 매개변수 할당식의 형태로써 매개변수의 이름이 중요하다. 심지어는 일부 매개변수는 생략될 수도 있으며, 그 순서도 무의미하다.

이런 차이가 나는 것은 부수효과의 허용과 관련이 있다. 함수는 동일한 입력에 대해서 항상 같은 값의 산출을 보장하지만, 함수블록은 전역변수가 허용되며 인스턴스마다 개별적인 지역변수를 가짐으로 함수블록의 호출 종료 후에도 그 값이 유지되는 것이 보장된다. 때문에 호출하는 쪽에서 구조적 변수를 다루듯이 그 값을 설정하거나 읽어 올 때 그 값이 보장 될 수 있다. 본 논문의 간략화된 구문구조가 출력 매개변수를 생략하고 있는 것도 이러한 방법으로 대체할 수 있기 때문이다.

이 특징은 또한 호출이 완료되는 시점에서도 차이를 유발한다. 함수의 실행결과는 누산기에 남지만 함수블록의 실행은 누산기를 변화시키지 않으며 출력변수를 통해서만, 그 결과를 준다.

이렇게 함수와 함수블록의 매개변수 전달의 차이는 함수 호출에서 이미 언급한 바와 같이 환경의 구별의 필요성을 야기한다. 함수 호출시에 *fun* 환경을 통해 e_{fp}를 얻어내는 것과 달리, 함수블록 호출시에는 *fb* 환경을 통해 e_{fbp}와 e_{fbpk}를 얻어낸다. e_{fbp}는 매개변수 이름으로부터 값이 전달될 주소를 알려주는 환경이며, e_{fbpk}는 매개변수의 입출력 유형을 알려주는 환경이다.

입출력 유형 환경은 원래는 접근 제어를 위해 사용되는 것으로 타입 검사 단계에서 사용되야 하지만, IL에서는 매개변수가 **VAR_IN_OUT**으로 선언된 경우 참조 변수로 다루어야 하는 또 다른 의미를 지니게 때문에 이 환경이 필요하다. 만약 이런 경우라면 매개변수로 값이 아닌 변수의 주소 -즉, *to_deep_loc*로 구해지는 주소-가 넘어가야 한다. 타입 시스템을 구성한다면 이 부분에 대해서도 검사해야 할 것이다.

CAL, RET 혹은 **JMP**에서는 **LDN**등과는 달리 **N** 이라는 접미사가 단독으로 쓰이지 않고, **C** 와 함께 사용되는데, 이는 앞쪽의 피연산자가 있는 다른 연산자들과는 다른 의미를 지닌다. **C** 의 의미는 누신기 값이 참인 경우에 효과가 있다는 의미이며, **CN** 은 반대로 거짓인 경우에 효과가 있다는 의미이다. 무효인 경우는 단순히 다음 프로그램 카운트로 넘어가기만 한다. 이들에 대한 정의는 조건 없는 연산인 [FB Call CAL], [JMP] 혹은 [RET]에 약간의 변형을 통해 쉽게 얻을 수 있으므로 여기에서는 생략한다.

복귀 연산자의 정의에서 중요한 점은 현재의 POU(callee)와 현재의 POU를 호출한 POU(caller)의 정보를 가져오기 위해 함수인지 함수블록인지를 구분하고 있다는 점이다. 이

[FB Call CAL]

$$(v,\ e,\ pc,\ (fun, fb)) \longrightarrow (?,\ e',\ pc',\ (fun, fb'))$$

where

$$\begin{cases} instr(pc) = \textbf{CAL}\ x_{callee}\ \{id_i := operand_i\}^{i\in 1..n} \\ e_v, mem \models x_{callee} \vdash callee\_id \\ e = ((e_v, caller\_id), (e_{sr}, mem)) \\ ((e'_v, pc', (e_{fbp}, e_{fbpk})), ret\_info) = fb\ callee\_id \\ fb' = fb[callee\_id \vdash ((e'_v, pc', (e_{fbp}, e_{fbpk})), (caller\_id, pc+1, e))] \\ \forall_{i\in 1..n} : vl_i = \\ \quad \begin{cases} to\_deep\_loc(e_v, mem, operand_i)\ \text{if}(e_{fbpk}(id_i) = \textbf{VAR\_IN\_OUT}) \\ to\_value(e_v, mem, operand_i)\ \text{otherwise} \end{cases} \\ \forall_{i\in 1..n} : l_i = e_{fbp}\ id_i \\ mem' = mem[\{l_i \vdash vl_i\}^{i\in 1..n}] \\ e' = ((e'_v, callee\_id), (e_{sr}, mem')) \end{cases}$$

(그림 12) 함수블록 호출의 의미구조

[RET Operation]

$$(v,\ e,\ pc,\ (fun, fb)) \longrightarrow (v',\ e',\ pc',\ (fun, fb))$$

where

$$\begin{cases} instr(pc) = \textbf{RET} \\ e = ((e_v, callee\_id), (e_{sr}, mem)) \\ (caller\_id, pc', v') = \\ \quad \begin{cases} (caller\_id, pc', v)\ \text{if}\ callee\_id \in \text{FUN\_ID} \\ \qquad \wedge (pou\_info, (caller\_id, pc')) = fun\ callee\_id \\ (caller\_id, pc', v')\ \text{if}\ callee\_id \in \text{FB\_INSTANCE\_ID} \\ \qquad \wedge (pou\_info, (caller\_id, pc', v')) = fb\ callee\_id \end{cases} \\ (e'_v, pc'', e_{fp}/fbp) = \\ \quad \begin{cases} pou\_info'\ \text{if}\ caller\_id \in \text{FUN\_ID} \\ \qquad \wedge (pou\_info', ret\_info') = fun\ caller\_id \\ pou\_info'\ \text{if}\ caller\_id \in \text{FB\_INSTANCE\_ID} \\ \qquad \wedge (pou\_info', ret\_info') = fb\ caller\_id \end{cases} \\ e' = ((e'_v, caller\_id), (e_{sr}, mem)) \end{cases}$$

(그림 13) 복귀문의 의미구조

에 따라 호출환경을 *fun*에서 가져와야 하는지 *fb*에서 가져와야 하는지가 차이가 난다.

이 후에는 간단히 앞서 호출문을 정의할 때, 수정해준 복귀 환경(*caller_id, pc',v'*)을 이용하여 간단히 복귀 할 수 있다. 복귀 환경은 현 POU의 식별자인 *callee_id*로부터 가져올 수 있다. 복귀 환경의 *caller_id*를 이용하여 변수환경 e_v를 복구하고 역시 복귀 환경의 프로그램 카운트 pc' 으로 다음 실행할 위치를 바꾸어 주면 호출 전의 환경에서 이어서 프로그램이 진행되게 된다.

누산기의 값은 현재 POU가 함수인지 함수블록인지에 따라 차이가 나는데, 함수의 경우는 현재 누산기의 값을 그대로 다음 누산기의 값으로 하며, 함수블록의 경우는 저장했던 호출 전 누산기의 값을 복구에 사용한다.

이로써 호출의 흔적은 누산기와 래치 환경 및 메모리 환경에만 남고 호출이 종료된다. **RETC**와 **RETCN**은 누산기의 값에 따라 복귀 여부가 차이가 있을 뿐 복귀 동작의 정의는 동일하므로 생략한다.

분기 연산은 단순히 프로그램 카운트만 바꾸어 주는 연산으로 (그림 14)와 같이 간단하게 정의한다. *lab*은 (그림 5)에서 가정한 것과 같이 *label*이 원래 선언되었던 위치의 명령어의 프로그램 카운트를 알려주는 문맥이다.

JMPC나 **JMPNC**와 같은 조건분기문의 정의 역시 [JMP Operation]을 약간 수정하여 정의할 수 있다.

[JMP Operation]

$$(v,\ e,\ pc,\ pou) \longrightarrow (v,\ e,\ pc',\ pou)$$

$$\text{where} \begin{cases} instr(pc) = \textbf{JMP}\ label \\ pc' = lab(label) \end{cases}$$

(그림 14) 분기문의 의미구조

6. 예 제

이해를 돕기 위해 앞에서 정의한 의미구조가 어떠한 방식으로 동작하게 되는지 간단한 예를 들어보겠다. <표 1>의 두 POU를 가정해 보자. 각 명령어 옆의 숫자는 해당 명령어의 프로그램 카운트를 의미한다.

이 POU들에 대한 변수 환경은 (그림 15)와 같이 가정될 수 있다. e_{v1} 과 e_{v2} 는 POU의 지역 변수를 위한 환경이며, e_{fp1} 과 e_{fp2} 는 해당 POU를 호출하기 위한 인터페이스 변수 환경이다. 또한 fun_0와 fb_0는 호출하기 위해 필요한 모든 정보를 갖는 POU 문맥이다.

이러한 가정에서 프로그램이 실행되는 예를 표로 나타내면 <표 2>와 같이 된다. 다른 부분은 사실 사소한 부분이며, 다소 복잡하게 정의되고 있는 함수 호출과 복귀문에 대해서 호출 환경이 어떻게 변화하게 되는지 주의 깊게 볼 필요가 있다.

〈표 1〉 예제 프로그램

호출 하는 쪽(*caller*)	호출 되는 쪽(*callee*)
FUNCTION Foo:INT VAR_INPUT Some:INT END_VAR 1 LD 10 2 Addtion 20 3 ST Some END_FUNCTION	FUNCTION Addition:INT VAR_INPUT In1: INT; In2: INT; END_VAR 10 LD In1 11 ADD In2 12 RET END_FUNCTION

〈표 2〉 예제 프로그램의 실행 예

$(7, e_0, 1, (fun_0, fb_0))$	**LD operation 적용** $(7, e_0, 1, (fun_0, fb_0)) \longrightarrow (10, e_0, 2, (fun_0, fb_0))$ where $\begin{cases} instr(1) = \textbf{LD 10} \\ e_0 = ((e_{v1}, \textbf{Foo}), (\emptyset, \emptyset)) \\ 10 = to\_value(e_{v1}, \emptyset, \textbf{10}) \end{cases}$
$(10, e_0, 2, (fun_0, fb_0))$	**Fun Call 적용** $(10, e_0, 2, (fun_0, fb_0)) \longrightarrow (7, e_1, 10, (fun_1, fb_0))$ where $\begin{cases} instr(2) = \textbf{Addition 20} \\ \textbf{Addition} \in \text{FUN\_ID} \\ e_0 = ((e_{v1}, \textbf{Foo}), (\emptyset, \emptyset)) \\ (e_{v2}, 10, e_{fp2}), (7, 2)) = fun\ \textbf{Addition} \\ fun_1 = fun_0[\textbf{Addition} \mapsto ((e_{v2}, 10, e_{fp2}), (\textbf{Foo}, 3))] \\ 20 = to\_value(e_{v1}, \emptyset, 20) \\ 200 = e_{fp2}\ 0 \\ mem_1 = \emptyset[200 \mapsto 20] \\ e_1 = ((e_{v2}, \textbf{Addition}), (\emptyset, mem_1)) \end{cases}$

〈표 3〉 예제 프로그램의 실행 예 (계속)

$(10, e_1, 10, (fun_1, fb_0))$	**LD operation 적용** $(7, e_1, 10, (fun_1, fb_0)) \longrightarrow (10, e_1, 11, (fun_1, fb_0))$ where $\begin{cases} instr(10) = \textbf{LD In1} \\ e_1 = ((e_{v2}, \textbf{Addition}), (\emptyset, mem_1)) \\ 10 = to\_value(e_{v2}, \emptyset, \textbf{In1}) \end{cases}$
$(10, e_1, 11, (fun_1, fb_0))$	**ADD operation 적용** $(10, e_1, 11, (fun_1, fb_0)) \longrightarrow (30, e_1, 12, (fun_1, fb_0))$ where $\begin{cases} instr(11) = \textbf{ADD In2} \\ e_1 = ((e_{v2}, \textbf{Addition}), (\emptyset, mem_1)) \\ 20 = to\_value(e_{v2}, \emptyset, \textbf{In2}) \end{cases}$
$(30, e_1, 12, (fun_1, fb_0))$	**RET 적용** $(30, e_1, 12, (fun_1, fb_0)) \longrightarrow (30, e_2, 3, (fun_1, fb_0))$ where $\begin{cases} instr(12) = \textbf{RET} \\ e_1 = ((e_{v2}, \textbf{Addition}), (\emptyset, mem_1)) \\ (\textbf{Foo}, 3) = \\ \begin{cases} (\textbf{Foo}, 3)\ \text{if}\ \textbf{Addition} \in \text{FUN\_ID} \\ \quad \wedge ((e_{v2}, 10, e_{fp2}), (\textbf{Foo}, 3)) = fun_1\ \textbf{Addition} \end{cases} \\ (e_{v1}, 1, e_{fp1}) = \\ \begin{cases} (e_{v1}, 1, e_{fp1})\ \text{if}\ \textbf{Foo} \in \text{FUN\_ID} \\ \quad \wedge ((e_{v1}, 1, e_{fp1}), (7, 2)) = fun_1\ \textbf{Foo} \end{cases} \\ e_2 = ((e_{v1}, \textbf{Foo}), (\emptyset, mem_1)) \end{cases}$
$(30, e_2, 3, (fun_1, fb_0))$	**ST operation 적용** $(30, e_2, 3, (fun_1, fb_0)) \longrightarrow (30, e_3, 4, (fun_1, fb_0))$ where $\begin{cases} instr(3) = \textbf{ST Some} \\ e_2 = ((e_{v1}, \textbf{Foo}), (\emptyset, \emptyset)) \\ e_{v1}, \emptyset = \textbf{Some} \mapsto 100 \\ e_3 = ((e_{v1}, \textbf{Foo}), (\emptyset, mem_1[100 \mapsto 30])) \end{cases}$

집합의 초기 상태:

$e_{v1} = \emptyset[\text{Some} \rightarrowtail 100]$

$e_{fp1} = \emptyset[0 \rightarrowtail 100]$

$e_{v2} = \emptyset[\text{In1} \rightarrowtail 200][\text{In2} \rightarrowtail 200]$

$e_{fp2} = \emptyset[0 \rightarrowtail 200][1 \rightarrowtail 201]$

$fun_0 = \emptyset[\text{Foo} \rightarrowtail ((e_{v1}, 1, e_{fp1}), (?, ?))][\text{Addition} \rightarrowtail (e_{v2}, 10, e_{fp2}), (?, ?))]$

$e_0 = ((e_{v1}, \text{Foo}), (\emptyset, \emptyset))$

$fb_0 = \emptyset$

FUN_ID = {Foo, Addition}

IL 기계의 초기 상태:

$(?, e_0, 1, (fun_0, fb_0))$

(그림 15) 집합과 기계의 초기 상태

7. 결 론

본 논문은 자동화 시스템이나 임베디드 시스템에서 사용하는 표준 제어 언어들 중에서 어셈블리어 수준의 IL 언어에 대하여 실행적 의미구조를 정형적으로 정의하였다. 이 정의를 이용하여 시뮬레이터를 구현하면 제어 프로그램을 PLC로 이식하지 않고도 테스트 할 수 있으며, 더 나아가서는 IL 언어를 대상으로 하는 다양한 프로그램 분석의 기반이 될 수 있다. 같은 목적을 가진 기존의 시도들은 IL 언어의 매우 작은 집합만을 대상으로 함으로써 실용적인 활용이 어렵다. 본 논문은 실용적인 프로그래밍에 필수적인 함수와 함수블록을 다룰 수 있도록 의미구조를 정의하였다.

추후 연구로는 본 논문의 결과를 이용하여 시뮬레이터를 구현하는 것 뿐만 아니라, 본 논문의 결과를 기반하여 프로그램 안전성을 검증하고 정적인 오류를 분석하는 다양한 시스템 즉, 타입 시스템, 요약 해석, 데이터 흐름 분석, 제어 흐름 분석 등을 고안할 수 있다.

참 고 문 헌

[1] H. Jack, Automating Manufacturing Systems with PLCs, http://claymore.engineer.gvsu.edu/~jackh/books.html, 2007

[2] IEC, International Standard IEC 61131-3 Programmable Controllers - Part 3: Programming Languages 2nd Edition, International Electrotechincal Commission, 2003.

[3] K. H. John, M. Tiegelkamp, IEC 61131-3:Programming Industrial Automation Systems, Springer Verlag Heidelberg, New York, 2001.

[4] R. Huuck, Software Verification for Programmable Logic Controllers, Ph.D Discussion, Christian-Albrechts-University of Kiel, 2003.

[5] R. Huuck, Semantics and Analysis of Instruction List Programs, Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages, Electronic Notes in Theoretical Computer Science 115, Elsevier, pp. 3-18, 2005.

[6] K. Slonneger, Executing an SECD machine using logic Programming, Proceedings of the Twenty-sixth Special Interest Group on Computer Science Education Technical Symposium on Computer Science Education, 1995.

[7] 노상훈, 신승철, IEC 61131-3 Instruction List의 의미구조: 함수확장. 한국기술교육대학교 PLLAB, TR-2007-03, http://pllab.kut.ac.kr/TR/2007/TR-2007-03.pdf, 2007.

[8] H. R. Nielson, F. Nielson, Semantics With Applications: A Formal Introduction, Wiley Professional Computing, 1992.

[9] G. D. Plotkin, A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[10] P. Bertelsen, Dynamic Semantics of Java Bytecode, Workshop on Principles of Abstract Machines, 1998.

[11] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[12] The Programming System CONTROL2000, PRAHM, http://www.prahm-ms.de/English/

[13] Programmable Logic Controllers Product Research, Reed Research Group and Control Engineering, 2006.

[14] E. Alonso, P. Kristofferson, J. McCann, Building Ambient Intelligence into a Ubiquitous Computing Management system, International Symposium of Santa Caterina on Challenges in the Internet and Interdisciplinary Research. SSCCII-2004, Amalfi, Italy, 2004.

신 승 철

e-mail : scshin@kut.ac.kr

1992년~1996년 인하대 전자계산학과 (박사)

1999년~2000년 캔사스주립대 연구원

1996년~2005년 동양대학교 컴퓨터공학부 부교수

2006년 ~ 현재 한국기술교육대학교 인터넷미디어공학부 조교수

관심분야 : 프로그래밍 언어, 소프트웨어 보안, 프로그램 분석 및 검증, 수리논리

노 상 훈

e-mail : noulne@kut.ac.kr

2001년~2005년 동양대학교
컴퓨터공학부 (학사)

2006년~현재 한국기술교육대학교
정보미디어공학과(석사과정)

관심분야 : 프로그래밍 언어, 프로그램 분석 및 검증