증명보조기 Coq을 이용한 래더 다이어그램 의미구조의 정형화

(Formalization of Ladder Diagram Semantics Using Coq)

신 승 철 †

(Seungcheol Shin)

요 약 산업자동화 분야에는 특수목적 마이크로콘트롤러인 PLC가 널리 사용된다. PLC 프로그램 분석과 검증을 위한 연구에서 우선적으로 해야 할 일은 PLC 프로그래밍 언어의 의미구조를 정형적으로 제시하는 것이다. 본 논문은 PLC 프로그래밍에 널리 사용하는 LD 언어의 의미구조를 정의한다. LD 언어는 그래픽 언어이기 때문에 먼저 텍스트 언어 Symbolic LD로 구문구조를 정형화한 다음에, Symbolic LD에 대한 의미구조를 정의할 수가 있다. 본 논문은 Symbolic LD의 의미구조를 자연 의미구조 기법으로 정의하고, 증명 보조기 Coq을 이용하여 정형화하였다.

키워드 : PLC, 래더다이어그램, 동작 의미구조, 증명보조기 Coq

Abstract Special-purpose microcontrollers PLCs have been widely used in the area of industrial automation. For the research of analysis and verification for PLC programs, first of all we have to specify formal sematics of PLC programming languages. This paper defines formally the operational semantics of LD language. After we transform the graphical language LD into its textual representation Symbolic LD, we give semantics of Symbolic LD since LD language is a graphical language. This paper defines the natural sematics of Symbolic LD and formalizes it in Coq proof assistant.

Key words : PLC, Ladder Diagram, Operational Semantics, Coq the Proof Assistant

1. 서 론

산업자동화 분야에서 자동 제어에 널리 사용되는 PLC (Programmable Logic Controller)는 특수목적 마이크로 콘트롤러로서 하드웨어 장치들과 설비들을 입력 접점과 출력 코일로 연결하여 제어한다. PLC의 국제표준 IEC61131은 PLC 자체에 대한 명세와 함께 PLC를 위한 프로그래밍 언어들을 제시한다. IEC61131은 PLC 프로그래밍 언어의 구문구조(syntax)와 의미구조(semantics)를 비정형적으로 기술하고 있다[1]. 여기서 제시된 5 가지 프로그래밍 언어는 그래픽 언어인 LD(Ladder Diagram), FBD(Function Block Diagram), SFC(Sequential Function Chart)와 텍스트 언어인 IL(Instruction List), ST(Structured Text)가 있다. 이 중에서 산업현장에서 가장 많이 사용되는 것은 LD와 IL이다.

PLC 프로그램 개발을 위한 개발환경이 상용제품과 공개 배포 형태로 제공되고 있다. 하지만 어떤 것도 PLC 언어의 정형적인 의미구조를 기반으로 하고 있지 않다. IEC61131이 비정형적인 명세만을 제공하고 있기 때문에 구현 중에 발생하는 많은 설계 쟁점이 존재하는데, 다양한 제품들이 나름의 선택을 함으로써 표준의 의도를 무색하게 한다. 또한 각 제품의 명세도 비정형적으로 정의되어, 설계자, 개발자, 사용자 사이에 명확한 의사 전달이 어렵기 때문에 개발 도구와 응용 소프트웨어의 신뢰성을 훼손할 수 있다. 또한 프로그램 분석과 검증에 대한 연구도 언어의 정형적인 명세를 기반으로 이루어지기 때문에, 최근에는 소프트웨어 개발 전 과정에 대한 정형화가 크게 중요시 되고 있다.

우리는 IEC61131 PLC 언어의 정형적인 의미구조에 기반하는 개발환경 휘모리(Whimori CDK)를 개발하고

· 본 연구는 부분적으로 소프트웨어무결점연구센터의 지원으로 이루어짐

† 종신회원 : 한국기술교육대학교 인터넷미디어공학부 교수
scshin@kut.ac.kr

논문접수 : 2009년 9월 14일
심사완료 : 2009년 10월 14일

있다[2]. PLC 프로그램 개발환경 휘모리는 그래픽 편집기, 컴파일러, 가상기계로 구성된 전형적인 개발환경에다가 모델검증 도구와 연역검증 도구를 장착한다. 검증 도구들은 PLC 프로그램의 정확성을 확인하기 위해서 주어진 명세를 만족하는지를 검증한다. 검증 도구는 물론이고 컴파일러와 가상기계도 PLC 언어의 정형적인 의미구조에 기반하여 개발하는 것이 휘모리의 특징이다.

본 논문은 PLC 언어 중 하나인 LD 언어에 대한 정형적인 의미구조를 정의한다. LD 프로그램의 정형적인 의미구조는 LD 언어 컴파일러에 대한 정형적 검증을 가능하게 하고, LD 프로그램 시뮬레이터(가상기계)의 구현에 기반이 된다. 본 논문은 LD 언어의 정형적인 의미구조를 자연 의미구조법(natural semantics)이라고 불리는 동작 의미구조 기법(operational semantics)을 이용하여 정의한다. 또한 정의된 LD의 의미구조를 증명 보조기(proof assistant) Coq에서 정형화하고, Coq을 이용하여 LD의 의미구조가 갖는 특성을 증명하는 예를 보여준다.

논문의 구성은 먼저 그래픽 언어 LD에 대응하는 Symbolic LD의 구문구조를 제시하고, Symbolic LD의 자연 의미구조를 단계별로 정의한다. 그 다음에 정의된 의미구조가 Coq에서 작성된 형태를 보여주고, 의미구조의 특성을 증명한 예를 보여준다.

2. 관련연구

PLC 언어들에 대한 정형적인 의미구조를 정의하는 것은 주로 PLC 프로그램의 자동 검증을 위해서 시도되어 왔다. 대부분의 시도들은 각 PLC 언어 프로그램의 의미구조를 오토마타, 페트리 넷, 특정 모델 검증기의 모델 등으로 직관적으로 옮겨쓴 다음, 모델 검증기를 통해서 자동 검증을 수행한다[3-5]. 본 논문의 의미구조 표기법은 특정 모델을 이용하지 않고 범용 언어의 의미구조 표기법인 동작 의미구조 표기법을 사용하는 것이 이들과 다르다. 동작 의미구조 표기법과 같은 범용 언어 의미구조 표기법을 이용하여 PLC 프로그램의 의미구조를 정의한 다른 예는 Shin[6], Huuck[7], Luckschus[8] 등이 있지만 대상 언어가 IL과 SFC이다.

증명 보조기 Coq은 정형 증명 관리 시스템이라고 할 수 있다. 이것은 정형 언어를 이용하여 수학적인 정의와 정리, 실행가능한 알고리즘 등을 작성할 수 있고, 자동 검증이 가능한 증명을 반자동으로 개발할 수 있는 환경을 제공한다[9]. Coq을 이용하여 정형 시스템을 작성하면, 정형 시스템의 성질을 Coq에서 증명할 수 있고, 이 증명의 자동 검증이 가능하다. 프로그래밍 언어에 대한 의미구조를 Coq에서 작성하고 그 성질들의 증명을 개발하면 정형 검증이 가능한 변환기(컴파일러), 분석기, 검증기, 시뮬레이터(인터프리터) 등을 구현하는 것이 가능하다[10]. 본 논문은 이러한 목적을 위해서 LD 언어의 의미구조를 Coq에서 작성한다. 타이머가 포함된 PLC 프로그램을 Coq에서 모델로 만드는 시도가 있었지만[11] 아직까지 PLC 언어의 의미구조를 Coq에서 정형화하는 시도는 없었다.

3. Symbolic LD

LD 언어로 작성된 단위 프로그램은 POU(Program Organization Unit)라고 하는데, 여러 개의 단(rung)으로 구성되고 각 단이 순차적으로 실행된다. POU의 실행은 LD 프로그램을 무한히 반복하는 루프이고, 한번의 반복을 스캔 싸이클이라고 한다. 각 스캔 싸이클은 세 단계로 이루어지는데, PLC 시스템에 연결된 센서들로부터 값을 읽어오는 입력 단계, LD 프로그램을 실행하는 계산 단계, 결과 값들을 장치 구동부로 전달하는 출력 단계이다. 본 논문에서는 주어진 LD 프로그램에 대하여 단일 스캔 싸이클의 계산 단계에만 집중한다.

그림 1은 간단한 LD 프로그램 예제를 보여주는데, 7개의 단으로 이루어진다. 각 단은 부분순서 관계를 이용하여 전단부(front)와 후단부(rear)로 나눌 수 있고, 그림 2와 같은 기호(텍스트) 표현을 생성할 수 있다[12]. 이때 LD 프로그램의 기호 표현은 그림 3의 추상 구문을 기반한다. 따라서 이제부터는 이 추상 구문으로 표현되는 LD 프로그램을 대상으로 설명한다.

그래픽 LD 프로그램의 각 단은 필요에 따라서 레이블을 붙일 수 있지만, 기호 LD 프로그램에서는 모든 단이 자연수 레이블을 가진다. 예제에서 switch들은 센서 접점을 의미하는 변수들이고 motor, warning, light들은 구동부를 의미하는 변수들이다.

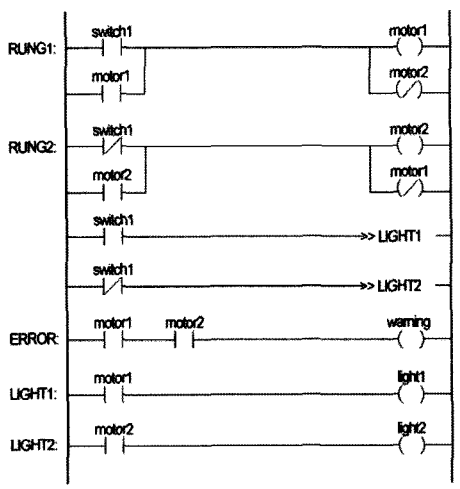

그림 1 간단한 LD 프로그램

추상 구문에서 하나의 LD 프로그램은 하나 이상의 단으로 구성되고, 각 단은 레이블, 전단부, 후단부로 구성된다. contact은 변수 입력을 의미하고 coil은 변수 출력을 나타낸다. contactn과 coiln은 각각 변수 입출력의 역(negated)을 나타낸다. 기호 & 와 || 는 각각 비트 연산 and와 or를 나타내고, 기호 ⋈ 는 단의 전단부와 후단부를 분리하는 기호이다. jumpto(i) 는 레이블 i의 단으로 실행 제어를 이동시킨다. 전단부는 입력과 and, or 연산으로 만들어지고, 후단부는 출력과 and, or 연산으로 만들어지는데, 후단부의 and는 전단부 형태와 후단부 형태를 연결한다는 것에 유의하자. 직관적으로 말해서 전단부는 출력부를 포함하지 않은 가장 큰(greatest lower bound) 부분식이고, 후단부는 출력부를 모두 포함하는 가장 작은(least upper bound) 부분식이다[12].

```
0:
contact(switch1) || contact(motor1) ⋈ coil(motor1) || coiln(motor2)
1:
contactn(switch1) || contact(motor2) ⋈ coil(motor2) || coiln(motor1)
2:
contact(switch1) ⋈ jumpto(5)
3:
contactn(switch1) ⋈ jumpto(6)
4:
contact(motor1) & contact(motor2) ⋈ coil(warning)
5:
contact(motor1) ⋈ coil(light1)
6:
contact(motor2) ⋈ coil(light2)
```

그림 2 그림 1의 기호 표현

$$
\begin{aligned}
\mathit{rungs} &::= \mathit{rung}\ \mathit{rungs} \mid \mathit{rung}\\
\mathit{rung} &::= \mathit{label} : \mathit{front} \bowtie \mathit{rear}\\
\mathit{front} &::= \mathit{front}\ \&\ \mathit{front} \mid \mathit{front} \parallel \mathit{front} \mid \mathit{input}\\
\mathit{rear} &::= \mathit{rear} \parallel \mathit{rear} \mid \mathit{front}\ \&\ \mathit{rear} \mid \mathit{output}\\
\mathit{input} &::= \mathrm{contact}(\mathit{var}) \mid \mathrm{contactn}(\mathit{var})\\
\mathit{output} &::= \mathrm{coil}(\mathit{var}) \mid \mathrm{coiln}(\mathit{var})\\
&\quad \mid \mathrm{jumpto}(\mathit{label}) \mid \mathrm{jumpton}(\mathit{label})
\end{aligned}
$$

그림 3 Symbolic LD의 추상구문

4. Symbolic LD의 의미구조

Symbolic LD 언어의 의미구조를 정의하기 전에, 먼저 구문구조를 구성하는 집합들을 소개한다. 모든 Symbolic LD 프로그램의 집합을 *LD*라고 하고 그 원소는 *rungs*라고 하자. *rungs*는 *rung*의 시퀀스이고 *rung*은 label: front ⋈ rear 로 표시되는 label, front, rear의 3쌍이다. front는 *Front*의 원소로서 전단부 형태를 가진 부분 단(partial rung)이고 rear는 *Rear*의 원소로서 후단부 형태의 부분 단이다.

LD 프로그램이 실행되는 시스템 상태(configuration) Γ는 다음과 같이 정의한다.

$$\Gamma = LD \times Bool \times Label \times Bool \times (Var \to Bool)$$

시스템 최종 상태 γ는 다음과 같이 따로 정의한다.

$$\gamma = Bool \times Label \times Bool \times (Var \to Bool)$$

그리고 주어진 LD 프로그램 *rungs*에 대한 실행 전이 관계 $\to_{LD} \subseteq \Gamma \times Y$는 다음과 같은 형태로 표기하고자 한다.

$$(rungs, v, l, j, e) \to_{LD} (v', l', j', e') \text{ where } \begin{cases} rungs \in LD \\ v, j, v', j' \in Bool \\ l, l' \in Label = \mathbb{N} \\ e, e' \in Var \to Bool \end{cases}$$

여기서 v와 v'은 연산이 일어나는 값들을 나타내는데, v는 이전 연산 결과값이고 v'은 이후 연산 결과값이다. 사실 이전 연산 결과값이 꼭 필요하지 않지만 시스템 상태의 모양을 동일하게 하기 위해서 포함한 것이다. l과 l'은 다음에 실행할 단을 가리키는 레이블이다. j와 j'은 분기가 발생했는지 여부를 나타낸다. e와 e'은 메모리 상태를 나타내는 함수이다.

실행 전이 관계 $\to_{LD}$를 추론 규칙을 이용하여 귀납적으로 정의하면 그림 4와 같다. 여기서 의미규칙 [*rungs*]는 하나 이상의 단을 가지는 LD 프로그램에 대한 것이고, [*rung*]은 단일 단의 의미규칙을 나타낸다.

$$\frac{(rung, v, l, j, e) \to_{LD} (v', l', j', e') \quad (rungs', v', l'+1, \mathbf{ff}, e') \to_{LD} (v'', l'', j'', e'')}{(rung\ \ rungs, v, l, j, e) \to_{LD} (v'', l'', j'', e'')}\ [rungs]$$

where $rungs' = jump(rungs, l')$

$$\frac{(front, v, l, j, e) \xrightarrow{l'} (v', l, j, e) \quad (rear, v', l, j, e) \xrightarrow{n} (v'', l', j', e')}{(label : front \bowtie rear, v, l, j, e) \to_{LD} (v'', l', j', e')}\ [rung]$$

그림 4 Symbolic LD의 의미구조 1/3

l은 현재 실행 중인 단의 다음 단을 가리키는 레이블로서 분기가 없다면 다음에 실행할 단을 가리킨다. l'은 현재 단을 실행하고 난 후에 결정되는데, 실제로 다음에 실행할 단을 가리킨다. $l = l'$ 이면 분기가 일어나지 않음을 나타낸다. j'는 이번 단에서 분기(jump)가 발생했는지 여부를 표시한다. 사실 j는 역할이 없지만 시스템 상태를 동일한 형태로 유지하기 위한 것이다. l'과 j'이 중복적인 정보를 주는 것처럼 보일 수 있지만, 사실 l'은 다음에 실행할 단으로의 분기를 위한 것이고, j'은 후단부에서 *jump(label)* || ***coil***(*var*) 와 같은 상황에서 분기가 일어날 때 ***coil***(*var*)의 실행을 무시하기 위한 것이다. 자세한 내용은 뒤에서 후단부의 의미규칙(그림 6)에서 분명해질 것이다.

하나의 단을 실행하고 나서 다음에 실행할 단을 가리키는 레이블 l 을 얻으면, 함수 *jump(rungs, l)*을 이용하여 *rungs*에서 일부만 발췌하여 실행을 계속한다. 발췌된 프로그램은 레이블 l의 단에서 마지막 단까지로 이루어진다.

$$jump(rungs, l) = \begin{cases} rungs & \text{if } l = label \\ jump(rungs', l) & \text{otherwise} \end{cases}$$

where $rungs = label : front \bowtie rear \;\; rungs'$

의미규칙 [*rung*]은 하나의 단이 실행될 때, 다음과 같이 표시되는 전단부와 후단부에 대한 실행 전이 $\xrightarrow{f}$와 $\xrightarrow{r}$를 각각 이용한다. 따라서 $\xrightarrow{f}$와 $\xrightarrow{r}$를 위한 각각의 의미규칙을 따로 정의한다.

$$\begin{array}{l}(front, v, l, j, e) \xrightarrow{f} (v', l', j', e') \\ (rear, v, l, j, e) \xrightarrow{r} (v', l', j', e')\end{array} \text{ where } \begin{cases} front \in Front \\ rear \in Rear \\ v, j, v', j' \in Bool \\ l, l' \in Label \\ e, e' \in Var \to Bool \end{cases}$$

전단부의 실행은 후단부에 비해 비교적 간단하다. 전단부의 실행은 *l, j, e*의 변화를 생성하지 않는다. 단지 연산값 *v*의 변화만 생성한다.

즉, **contact**(*var*)은 변수 값을 읽어들여 연산값에 기록한다. **contactn**(*var*)은 변수 값의 부정값을 기록한다. 귀납적으로 정의된 [*front-and*]와 [*front-or*]는 *front_1*의 결과와 *front_2*의 결과에 비트 곱과 합을 각각 취한다.

후단부의 실행은 **coil**과 **jumpto**의 처리가 주된 임무이다. ***coil***(*var*)과 ***coiln***(*var*)은 *v, l, j*의 값은 변화시키지 않고 메모리 상태 *e*의 변화만 생성한다. ***coil***(*var*)은 *v*의 값이 ***tt***이면 새로운 메모리 상태 *e'*에 *var*을 ***tt***로 갱신한다. *v*의 값이 ***tt***가 아니면 *var*을 ***ff***로 갱신한다. ***coiln***(*var*)은 각각 그 부정값으로 갱신한다.

jumpto(*label*)은 *v, e*는 변화시키지 않는다. *v*가 ***tt***이면 분기가 일어난다. 즉, 다음에 실행할 단을 가리키는 *l'*이 *label*이 되고, 분기 여부를 가리키는 플래그 *j'*은 ***tt***가 된다. *v*가 ***ff***이면 분기가 일어나지 않으므로 *l' = l* 이고 *j' = **ff***이다. ***jumpton***(*label*)은 반대로 *v*가 ***ff***일 때 분기가 일어나고 ***tt***이면 분기가 일어나지 않는다.

플래그 *j'*의 역할은 [*rear-or-jump*]나 [*rear-or-nojump*]에서 분명해지는데, *rear_1* || *rear_2* 에서 *rear_1*의 실행이 분기를 일으키면([*rear-or-jump*]에서 *j' = **tt***), *rear_2*의 실행은 건너뛴다. 반면에 *rear_1*의 실행이 분기를 일으키지 않으면([*rear-or-nojump*]에서 *j' = **ff***), *rear_1*의 실행 결과 *e'*를 가지고 *rear_2*를 실행하여 최종 메모리 상태 *e''*을 얻는다.

$$\frac{(front_1, v, l, j, e) \xrightarrow{f} (v', l, j, e) \quad (front_2, v, l, j, e) \xrightarrow{f} (v'', l, j, e)}{(front_1 \;\&\; front_2, v, l, j, e) \xrightarrow{f} (v' \wedge v'', l, j, e)}\ [front-and]$$

$$\frac{(front_1, v, l, j, e) \xrightarrow{f} (v', l, j, e) \quad (front_2, v, l, j, e) \xrightarrow{f} (v'', l, j, e)}{(front_1 \,\|\, front_2, v, l, j, e) \xrightarrow{f} (v' \vee v'', l, j, e)}\ [front-or]$$

$$\frac{}{(\mathbf{contact}(var), v, l, j, e) \xrightarrow{f} (e\ var, l, j, e)}\ [contact]$$

$$\frac{}{(\mathbf{contactn}(var), v, l, j, e) \xrightarrow{f} (\neg(e\ var), l, j, e)}\ [contact-negated]$$

그림 5 Symbolic LD의 의미구조 2/3

5. Coq 정형화

우리는 지금까지 정의한 LD 언어의 의미구조를 증명보조기 Coq을 이용하여 정형화하였다. 결과물인 Coq 코드는 [13]에서 볼 수 있다. 이를 이용하면 LD 언어의 의미구조와 관련한 성질들을 증명하고 그 증명 자체도 자동으로 검증할 수 있다. 여기서는 LD의 의미구조와 관련하여 증명된 몇 가지 성질들을 예로 보여준다.

$\rightarrow_{LD}$를 정의할 때, 분기를 표현하기 위해서 프로그램의 일부를 발췌하는 함수 *jump*를 생각해보자. 함수 *jump*를 취하면 본래의 프로그램 길이보다 작거나 같은 프로그램을 발췌한다. 다시 말해서, *jump(rs, l) = rs'* 라고 하면 $|rs| \leq |rs'|$이다. 이 사실은 다음의 정리로 정형화되어 증명되었다. 여기서 **LDjump**가 함수 *jump*를 Coq에서 정형화한 것이다.

```
Fixpoint LDjump (s:Rungs)(l:Label){struct s} :
Rungs := match s with
  | nil => nil
  | (l', (f, r)) :: rungs' =>
    if (label_eq_dec l l') then s
    else LDjump rungs' l
  end.

Lemma jumpforward : forall (s: list Rung) (ll:Label),
                    (length (LDjump s ll)) <= (length s).
```

따라서 레이블 *label*이 붙은 단에서 ***jumpto***(*l*)이 실행될 때, *label* ≠ *l*이기만 하면 프로그램의 종료를 보장할 수 있다.

또한 $\xrightarrow{f}$의 실행은 시스템 상태에서 *l, j, e* 의 변화를 일으키지 않는다. 사실 시스템 상태에 변화되지 않는 인자를 포함한 이유는 추후에 다른 언어 실행 모델의 시스템 상태와 동일하게 하기 위한 것이지만 이 논문에서는 특별히 언급되지 않았다. 어쨌든 이런 사실은 다음의 정리들로 정형화되고 증명되었다.

```
Fixpoint fronteval' (f:Front)(c:ldconfig){struct f} :
ldconfig := match f with
  | Contact var => ((lds c) var, ldl c, ldj c, lds c)
  | Contactn var => (negv ((lds c) var), ldl c, ldj c, lds c)
```

$$\frac{(front, v, l, j, e) \xrightarrow{f} (v', l, j, e) \quad (rear, v', l, j, e) \xrightarrow{r} (v'', l', j', e')}{(front \,\&\, rear, v, l, j, e) \xrightarrow{r} (v'', l', j', e')}\ [rear - and]$$

$$\frac{(rear_1, v, l, j, e) \xrightarrow{r} (v', l', \mathbf{tt}, e')}{(rear_1 \,\|\, rear_2, v, l, j, e) \xrightarrow{r} (v', l', \mathbf{tt}, e')}\ [rear - or - jump]$$

$$\frac{(rear_1, v, l, j, e) \xrightarrow{r} (v', l', \mathbf{ff}, e') \quad (rear_2, v, l, j, e') \xrightarrow{r} (v'', l'', j'', e'')}{(rear_1 \,\|\, rear_2, v, l, j, e) \xrightarrow{r} (v'', l'', j'', e'')}\ [rear - or - nojump]$$

$$\frac{}{(\mathbf{coil}(var), v, l, j, e) \xrightarrow{r} (v, l, j, e')}\ [coil]$$

where $\begin{cases} e' = e[var \mapsto \mathbf{tt}] & \text{if } v = \mathbf{tt} \\ e' = e[var \mapsto \mathbf{ff}] & \text{otherwise} \end{cases}$

$$\frac{}{(\mathbf{coiln}(var), v, l, j, e) \xrightarrow{r} (v, l, j, e')}\ [coil - negated]$$

where $\begin{cases} e' = e[var \mapsto \mathbf{ff}] & \text{if } v = \mathbf{tt} \\ e' = e[var \mapsto \mathbf{tt}] & \text{otherwise} \end{cases}$

$$\frac{}{(\mathbf{jumpto}(label), v, l, j, e) \xrightarrow{r} (v, l', j', e)}\ [jumpto]$$

where $\begin{cases} l' = label \text{ and } j' = \mathbf{tt} & \text{if } v = \mathbf{tt} \\ l' = l \text{ and } j' = \mathbf{ff} & \text{otherwise} \end{cases}$

$$\frac{}{(\mathbf{jumpton}(label), v, l, j, e) \xrightarrow{r} (v, l', j', e)}\ [jumpto - negated]$$

where $\begin{cases} l' = label \text{ and } j' = \mathbf{tt} & \text{if } v = \mathbf{ff} \\ l' = l \text{ and } j' = \mathbf{ff} & \text{otherwise} \end{cases}$

그림 6 Symbolic LD의 의미구조 3/3

```
| Fand f1 f2 => fronteval' f2 (fronteval' f1 c)
| For f1 f2 => ldorv (fronteval' f1 c) (fronteval' f2 c)
end.

Lemma no_ch_label : forall (f:Front)(c:ldconfig),
                    ldl (fronteval' f c) = ldl c.
Lemma no_ch_jump : forall (f:Front)(c:ldconfig),
                   ldj (fronteval' f c) = ldj c.
Lemma no_ch_state : forall (f:Front)(c:ldconfig),
                    lds (fronteval' f c) = lds c.
```

여기서 **fronteval'**은 $\xrightarrow{f}$를 함수로 구현한 것이다. Coq 구현에는 관계로 구현한 **fronteval**도 있지만 본 논문의 예제에서는 함수 버전을 사용한다. 또한 **ldl, ldj, lds**는 각각 시스템 상태에서 레이블, 분기여부, 상태함수을 발췌하는 함수들이고, 여기에 연산값을 덧붙여 4개의 필드로 이루어진 시스템 상태는 **ldconfig** 타입으로 표현된다. 마지막에 **ldorv**는 두 시스템 상태에서 연산값만 발췌하여 논리합 연산을 취하는 함수이다. 정리 **no_ch_label**, **no_ch_jump**, **no_ch_state**는 각각 **fronteval'**의 수행 결과가 레이블, 분기, 상태함수는 변경하지 않는다는 것을 나타낸다.

6. 결 론

본 논문은 산업자동화에 사용되는 PLC 언어 LD의 정형적인 의미구조를 정의하고 증명 보조기 Coq에서 정형화한 다음에, 관련 성질을 증명하는 예를 보여주었다. PLC 언어들은 구문구조와 의미구조의 설계시에 정형 설계 기법을 사용하지 않아서 엄밀한 정의가 존재하지 않으므로 구현된 시스템마다 조금씩 다른 의미를 가지게 된다. 정형적인 의미구조를 엄밀하게 정의하면 검증된 컴파일러와 시뮬레이터의 구현이 가능하다.

휘모리 개발환경의 요소들은 모두 정형적인 의미구조를 기반으로 개발되기 때문에, PLC 프로그램의 검증 도구들을 정형적으로 개발하는 데에도 유용하다. 휘모리 개발 환경에 장착하는 PLC 프로그램 검증 도구는 모델 검증기와 정리 증명기가 모두 고려되고 있다. 이때에 모델 검증기를 위한 모델 생성이나 정리 증명기를 위한 프로그램 변환 등은 본 논문의 결과를 포함하여 정형적인 의미구조를 기반으로 설계되고 있으며, 필요한 주요 성질들의 증명도 또한 Coq을 이용하여 자동 검증될 예정이다.

참 고 문 헌

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

[2] S. Shin, M. Kwon, and S. Rho, Whimori CDK: a Control Program Development Kit, The International Conference of COMPUTING in Engineering, Science and Informatics, 2009.

[3] M. Bani Younis and G. Frey. Formalization of existing PLC Programs: A Survey Proceedings of CESA 2003 paper no. S2-R-00-0239, 2003.

[4] O. Rossi, Ph. Schnoebelen. Formal Modelling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems pp.177-182, 2000.

[5] I. Hatono, K. Baba, M. Umano, H. Tamura. Automatic Generation of Fault Detection Models for Programmable Constroller-Based Manufatureing Systems Using Complementary-Places Petri Nets IFAC World Congress, 1996.

[6] S. Shin and S. Roh. Operational Semantics for Instruction List with Functions Journal of Korea Informaion Processiong Society vol.14-A, no.7, 2007 (in Korean)

[7] Ralf Huuck. Software Verification for Programmable Logic Controllers Ph.D dissertation, Christian-Albrechts-Universität zu Kiel, 2003.

[8] Ben Lukoschus. Compositional Cerification of Industrial Control Systems Ph.D dissertation, Christian-Albrechts-Universität zu Kiel, 2005.

[9] Y. Bertot and P. Casteran. Interactive theorem proving and program development: Coq'Art : the calculus of inductive constructions Texts in theo-

retical computer science, Springer, 2004.
[10] Y. Bertot. Theorem proving support in programming language semantics Research report no.6242, INRIA Sophia Antipous, 2007.
[11] H. Wan, G. Chen, X. Song and M. Gu. Formalization and Verification of PLC Timers in Coq 33rd Annual IEEE International Computer Software and Applications Conference pp.315-323, 2009.
[12] M. Kwon, S. Shin. Translating Ladder Diagrams into Instruction List Using Partial Order Relation KIISE KCC, 2008 (in Korean)
[13] Coq scirpit for Formalization of Ladder Diagram Semantics, http://pllab.kut.ac.kr/Coq/ldsemantics/ldsemantics.html

신 승 철
1992년~1996년 인하대 전자계산학과(박사). 1999년~2000년 캔사스 주립대 연구원. 1996년~2006년 동양대 컴퓨터학부 부교수. 2006년~현재 한국기술교육대 인터넷미디어공학부 조교수. 관심분야는 프로그래밍 언어, 프로그램 분석 및 검증, 소프트웨어 보안, 수리논리