비혼 논리 프로그램의 효율적 수행

신동하* · 백윤철*

An Efficient Execution of Non-Horn Logic Programs

Dongha Shin* · Yun Cheol Baek*

요 약

비혼(non-Horn) 논리 프로그램은 혼(Horn) 논리 프로그램을 1계 술어 논리(1st order predicate logic) 수준으로 확장하였기 때문에 표현력은 크지만 효율적으로 구현된 사례가 없어서 실용적인 언어로 사용되지는 못하였다. 지금까지 연구된 효율적인 방법은 비혼 논리 프로그램을 증명절차 InH-Prolog의 의미를 이용하여 동등한 혼 논리 프로그램으로 변환한 후 변환된 혼 논리 프로그램을 WAM(Warren Abstract Machine) 명령어로 컴파일하여 수행시키는 방법이다. 본 논문에서는 이 방법을 향상시키기 위하여 비혼 논리 프로그램을 효율적으로 수행하는 EWAM(Extended WAM)과 비혼 논리 프로그램을 EWAM 명령어로 컴파일하는 방법을 제안한다. 또한 본 논문에서는 제안한 EWAM의 에뮬레이터 및 컴파일러를 구현하여 그 성능을 측정하였다. 본 논문에서 구현한 EWAM 에뮬레이터 및 컴파일러의 성능을 측정한 결과 기존 방법보다 매우 효율적임을 확인하였다.

ABSTRACT

Non-Horn logic programs are extended from Horn logic programs to the level of 1st order predicate logic. Even though they are more expressive than Horn logic programs, they are not practically used because we do not have efficient implementations. Currently to execute non-Horn logic programs, we translate them to equivalent Horn logic programs using the proof procedure InH-Prolog and compile the Horn logic programs to WAM(Warren Abstract Machine) instructions. In this paper, we propose EWAM(Extended Warren Abstract Machine) that executes non-Horn logic programs more efficiently and a compilation scheme that compiles non-Horn logic programs to the EWAM instructions. We implement an EWAM emulator and a compiler and measured the performance of the EWAM emulator and the compiler and found that they are very efficient.

키워드

비혼(non-Horn) 논리 프로그램, 혼(Horn) 논리 프로그램, 증명 절차(proof procedure), WAM, 논리 언어 컴파일러

Ⅰ. 서 론

혼 논리(Horn logic) 언어[1]는 증명 절차(proof procedure) SLD-도출(SLD-resolution)을 좌에서 우로(left-to-right) 깊이 우선 탐색(depth-first search) 방법[2]을 사용하며 구현한다. 혼 논리 언어의 효율적인 구현은 1983년 D. H. D. Warren이 제안한 WAM(Warren Abstract Machine)을 사용한 컴파일 방법[3][4]으로서 Prolog 언어와 같은 대부분의 혼 논리 언어는 WAM으로 구현되어 있다.

비혼 논리(non-Horn logic) 언어[5]는 혼 논리 언어를 1계 술어 논리(1st order predicate logic) 수준으로 확장

* 상명대학교 소프트웨어학부 부교수

접수일자 : 2005. 1. 24

한 언어로서 선언적 의미(declarative semantics)와 증명 절차(proof procedure)에 대한 연구가 이루어졌다[6][7]. 비혼 논리 언어에 완전한(complete) 증명 절차로 SLI-도출(SLI-resolution)[5], MPRF (Modified Problem Reduction Format)[8], InH-Prolog(Inheritance near-Horn Prolog)[9][10] [11][12] 등이 제안되었으며 이들 중 InH-Prolog가 언어 구현 시 제어의 흐름이 자연스럽고 효율적임이 연구되었다[13]. 비혼 논리 언어는 혼 논리 언어와는 달리 효율적인 구현이 없어서 실용적으로 사용되지는 못하였다. 증명 절차 InH-Prolog를 창안한 D. Loveland 연구 팀이 증명 절차 InH-Prolog의 전신인 nH-Prolog를 제안할 때 개발한 해석기 (interpretation) 방법[14]이 최초로 구현되었지만 이 방법은 프로그램을 컴파일하지 않고 증명 절차의 의미를 사용하여 해석하여 수행하기 때문에 매우 비효율적이어서 사실상 실용적으로 사용하기는 불가능하였다. 그 후 1999년 이 방법보다 효율적인 컴파일(compilation) 방법이 제안되어 구현되었다[15]. 이 방법은 해석기 방법보다는 효율적이지만 최종 생성되는 명령어가 혼 논리 언어용으로 제안된 WAM 명령어이어서 비혼 논리 프로그램을 최대한 효율적으로 수행하지는 못하였다.

본 논문에서는 기 제안된 컴파일 방법을 더 향상시키는 방법을 제안하고 구현한다. 본 논문은 기존 방법이 WAM으로의 컴파일 방법이여서 근원적으로 비혼 논리 프로그램에 최적화되지 않는 부분이 존재하기 때문에 기존 WAM을 확장한 EWAM (Extended Warren Abstract Machine)을 제안하고 비혼 논리 프로그램을 EWAM 명령어로 컴파일하는 방법을 제시한다. 본 논문에서 제안한 EWAM의 에뮬레이터 및 컴파일러를 실제 구현하여 시험한 결과 기존의 방법에 비하여 매우 효율적이라는 결과를 얻었다. 본 논문의 2장에서는 본 논문과 연관된 주요 연구 결과에 관하여 기술하고 본 연구와의 관계를 설명한다. 3장에서는 본 논문이 제안하는 EWAM 및 EWAM으로의 컴파일 방법에 대하여 설명하고, 4장에서는 EWAM 에뮬레이터 및 컴파일러의 실제 구현에 관하여 기술한다. 5장에서는 구현된 EWAM 및 컴파일러의 성능 측정 결과를 기술하고 6장에서는 본 논문의 의의를 설명한다. 본 논문에서는 참고문헌 [2][5]에서 사용하는 논리 프로그래밍 관련 용어 및 정의를 대부분 재 정의하지 않고 사용한다.

Ⅱ. 관련 연구

본 장에서는 본 논문 주제와 직접적으로 관련된 연구에 대하여 기술하고 본 논문과 관계를 설명한다.

2.1. 증명 절차 InH-Prolog

본 논문에서 제안하는 EWAM 및 컴파일러 연구에 이용하는 증명 절차인 InH-Prolog는 1992년 D. W. Loveland가 고안한 비혼 논리 프로그램에 안전하고(sound) 완전한(complete) 증명 절차인 증명 절차이다[12]. 증명 절차 InH-Prolog는 주어진 비혼 논리 프로그램 LP와 목표 클로즈(goal clause) G가 있을 때 PCONTRA(LP∪{G})를 사용하여 증명(proof)을 구한다. 여기서 LP는 비혼 클로즈의 집합이고 PCONTRA(S)는 주어진 비혼 클로즈 집합 S의 양의 대우 클로즈(positive contrapositive) 집합이다.

증명 절차 InH-Prolog은 아래와 같은 클로즈 축약 규칙(clause reduction rule), 조상 삭제 공리(ancestor cancellation axiom) 및 재시작 규칙(restart rule)으로 구성된다.

- 클로즈 축약 규칙(C): 만약 "H :- B1, ···, Bn" ∈ PCONTRA(LP∪{G})이면 다음과 같은 규칙을 사용할 수 있다. 여기서 Γ는 리터럴(literal)의 집합이다.

$$\frac{\Gamma \rightarrow B1 \cdots \Gamma \rightarrow Bn}{\Gamma \rightarrow H}\ C$$

- 조상 삭제 공리: 만약 L ∈ Γ이면 다음과 같은 공리를 사용할 수 있다.

$$\Gamma \rightarrow L$$

- 재시작 규칙(R): 만약 L이 음(negative)의 리터럴이면 다음과 같은 재시작(restart) 규칙을 사용할 수 있다.

$$\frac{\Gamma \cup \{not\ L\} \rightarrow false}{\Gamma \rightarrow L}\ R$$

하나의 증명(proof)은 이들 규칙을 귀납적으로 사용하여 시퀀트(sequent) "{} → false"가 존재함을 보이는 것이다.

2.2. 해석기 방법의 구현

Loveland 연구팀은 증명 절차 InH-Prolog의 전신인 증명 절차 nH-Prolog를 발표한 후 이 증명 절차를 이용한 비혼 논리 언어의 해석기(interpreter)를 구현하였다[14]. 이 해석기는 비혼 논리 프로그램을 데이터처럼 저장하고 해석기가 각 저장된 프로그램을 읽어가면서 수행하는 전통적인 해석기이다. 이 방법의 문제점은 해석기 방법의 근원적인 한계인 수행 속도가 매우 느리다는 점이다. 이 해석기는 실험실 수준의 해석기로서 실제 응용에 사용되기는 힘들다는 단점도 있다.

2.3. WAM 컴파일 방법의 구현

Loveland 연구팀의 구현 방법을 개선한 컴파일 방법이 1999년 제안되었다[15]. 이 방법은 해석기 방법과는 달리 주어진 비혼 논리 프로그램을 증명 절차 InH-Prolog의 의미에 따라 혼 논리 프로그램으로 변환하고 변환된 혼 논리 프로그램을 WAM 명령어로 컴파일하여 수행하는 방법이다. 이 논문에서 저자는 "단순 변환" 방법과 "효율 변환" 방법을 제안하였다. 단순 변환은 증명 절차의 규칙과 공리를 직접 혼 클로즈로 변환하는 방법이고, 효율 변환은 단순 변환된 프로그램이 보다 효율적으로 수행시키기 위하여 입력 비혼 논리 프로그램에서 사용되는 술어 이름을 변환된 혼 논리 프로그램에서도 그대로 사용할 수 있게 변환하여 수행 속도를 빠르게 구현한 방법이다.

● 단순 변환 방법

비혼 논리 프로그램을 혼 논리 프로그램으로 변환하는 기본 아이디어는 클로즈 축약 규칙, 조상 삭제 공리 및 재시작 규칙을 혼 클로즈로 표현하는 것으로 변환된 혼 논리 프로그램 전체가 하나의 술어 이름 'seq/2'를 가진다. 여기서 증명 절차 InH-Prolog를 구성하는 시퀀트 "Γ → H"는 "seq(G,H)"로 변환 표현된다. 다음은 단순 변환의 변환 규칙(transformation rule)이다.

- 클로즈 축약 규칙의 변환: 만약 "H :- B1, ⋯, Bn" ∈ PCONTRA(LP∪{G})이면 이 클로즈는 다음과 같은 혼 클로즈로 변환한다.

```
seq(G, H) :- seq(G, B1), ⋯, seq(G, Bn).
```

- 조상 삭제 공리의 변환: 증명 절차 InH-Prolog의 조상 삭제 공리를 위하여 다음과 같은 혼 클로즈가 필요하다. 여기서 술어 'member'는 Prolog 언어 등에서 정의된 술어 'member'와 같은 의미이다.

```
seq(G, L) :- member(L, G).
```

- 재시작 규칙의 변환: 증명 절차 InH-Prolog의 재시작 규칙은 다음과 같은 혼 클로즈를 필요로 한다.

```
seq(G, not L) :- seq([L|G], false).
```

보기 1 (단순 변환) 다음과 같은 비혼 논리 프로그램 LP와 목표 클로즈 G가 있을 경우 단순 변환의 과정은 다음과 같다.

LP:
```
(1) on(a,b).
(2) on(b,c).
(3) color(a,green).
(4) color(c,blue).
(5) color(b,blue); color(b,green).
```
G:
```
(6) false :- on(X,Y),
        color(X,green), color(Y,blue).
```

PCONTRA(LP∪{G})를 구하면 다음과 같다.

```
(1) on(a,b).
(2) on(b,c).
(3) color(a,green).
(4) color(c,blue).
(5-1) color(b,blue) :- not color(b,green).
(5-2) color(b,green) :- not color(b,blue).
(6) false :- on(X,Y),
        color(X,green), color(Y,blue).
```

증명 절차 InH-Prolog를 이용하여 위 비혼 논리 프로그램을 혼 논리 프로그램으로 변환한 단순 변환은 다음과 같다. 아래에서 시퀀트의 Γ를 표현한 변수 G는 리스트이므로 리스트의 표현 기호인 '['와 ']'를 사용하여 표현되기도 한다. 또 클로즈 중 (a)는 조상 삭제 공리를 표현한 혼 클로즈이고, (r)은 재시작 규칙을 표현한 혼 클로즈이다. 다음은 단순 변환 결과이다.

```
(a) seq(G,L) :- member(L,G).
(r) seq(G, not L) :- seq([L|G], false).
(1) seq(G,on(a,b)).
(2) seq(G,on(b,c)).
(3) seq(G,color(a,green)).
(4) seq(G,color(c,blue)).
```

```
(5-1) seq(G,color(b,blue)) :-
        seq(G,not color(b,green)).
(5-2) seq(G,color(b,green)) :-
        seq(G,not color(b,blue)).
(6) seq(G,false) :-
        seq(G,on(X,Y)),
        seq(G,color(X,green)),
        seq(G,color(Y,blue)).
```

● 효율 변환 방법

앞의 단순 변환 방법은 간단하고 정확한 변환 방법이지만 다음과 같은 단점이 있다. 첫째, 변환된 혼 논리 프로그램 전체가 하나의 술어 이름 'seq'를 가지기 때문에 수행 시 비효율적이다. 즉 특별한 최적화를 하지 않는 이상 술어 'seq/2'를 부르기 위하여 더 많은 횟수의 일치화를 하게 된다. 둘째, 변환된 혼 논리 프로그램은 입력 비혼 논리 프로그램에 등장하는 술어 이름을 전혀 사용하지 않기 때문에 변환된 프로그램을 이해하거나 디버깅할 때 매우 어려움이 있다. 이를 해결하기 위하여 술어 이름 'seq'를 사용하지 않고 입력 비혼 논리 프로그램에 등장하는 술어 이름을 직접 사용하는 효율적인 변환 방법이 제시되었다. 다음은 증명 절차 단순 변환의 결과를 효율 변환하는 방법이다.

- 클로즈 축약 규칙의 변환: 단순 변환의 혼 클로즈인 "seq(G1, H) :- seq(G2, B1), ···, seq(G2, Bn)"는 다음과 같은 혼 클로즈로 다시 표현될 수 있다. 여기서 입력 프로그램에서 술어 H의 인수의 수(arity)가 n이면 효율적으로 변환된 프로그램에서 술어 H의 arity는 (n+1)이다.

```
H(···, G1) :- B1(···, G2), ···, Bn(···, G2).
```

- 조상 삭제 공리의 변환: 단순 변환의 조상 삭제 공리를 구현하기 위한 혼 클로즈는 다음과 같은 혼 클로즈로 다시 표현된다. 증명 절차 InH-Prolog는 PCONTRA(LP∪{G})를 사용하기 때문에 비혼 클로즈를 만드는 술어 이름에 대해서만 조상 삭제 공리가 필요하다. 이는 비혼 클로즈를 만드는 술어 이름들만 시퀀트의 Γ에 수집되기 때문이다.

```
L(···, G) :- member(L(···), G).
```

- 재시작 규칙의 변환: 증명 절차 InH-Prolog의 재시작 규칙은 다음과 같은 혼 클로즈로 다시 표현될 수 있다.

```
not L(···, G) :- false([L(···)|G]).
```

보기 2 (효율 변환) 다음은 보기 1에서 단순 변환된 프로그램을 다시 효율 변환한 프로그램이다. 이 보기에서 비혼 클로즈를 만드는 술어는 'color'이므로 조상 삭제 공리는 아래 (a)와 같이 술어 'color'에 대하여 하나만 존재한다.

```
(a) color(A1,A2,G) :- member(color(A1,A2),G).
(r) not L(···,G) :- false([L(···)|G]).
(1) on(a,b,G).
(2) on(b,c,G).
(3) color(a,green,G).
(4) color(c,blue,G).
(5-1) color(b,blue,G):-not color(b,green,G).
(5-2) color(b,green,G):-not color(b,blue,G).
(6) false(G) :-
        on(X,Y,G),
        color(X,green,G),
        color(Y,blue,G).
```

Ⅲ. EWAM을 사용한 컴파일 방법

3.1. EWAM의 구조

앞 장에서 설명한 효율 변환에서 성능과 관련된 가장 큰 문제는 변환된 프로그램의 모든 술어에 G라는 인수가 추가로 포함되어 있다는 것이다. 이 추가 인수 때문에 프로그램 수행 시 일어나는 일치화(unification) 시간이 더 길어질 수밖에 없다. 증명 절차 InH-Prolog가 비혼 논리 프로그램이 아닌 혼 논리 프로그램에 적용될 때 인수 G의 값이 항상 공집합임을 고려하면 비혼 논리 프로그램에서 인수 G의 사용은 필연적이라고 할 수 있다. 즉 인수 G의 제거는 비혼 논리 프로그램을 사용할 경우 근원적으로 불가능하다. 하지만 모든 술어에 같은 인수 G가 포함되어 있기 때문에 WAM 내부에서 이를 처리하는 메커니즘만 있다면 훨씬 효율적인 구현이 가능하다. 현재 혼 논리 프로그램을 컴파일하기 위하여 제안된 WAM은 추가 인수 G를 특별히 저장하거나 처리하는 부분이 포함되어 있지 않기 때문에 앞에서 제안된 효율 변환에서는 추가 인수 사용 이외의 방법으로 구현을 하기는 힘들다.

이러한 문제를 근원적으로 해결하는 방법은 WAM을 확장하여 인수 G를 효율적으로 처리하는 것이다. 본 논문에서 확장하는 EWAM은 인수 G를 인수로 처리하지 않고 EWAM에서 새로 정의하는 레지스터로

처리하여 술어 부름(predicate call) 시 일어나는 일치화 속도를 증가시킨다. 또한 새로운 레지스터의 등장으로 G의 내부 데이터 처리를 EWAM 내부에서 할 수 있는 명령어의 추가가 필요하다. 이를 위하여 새로운 리터럴을 G에 추가하는 명령어 'gamma_add' 및 G에 주어진 리터럴이 포함되어 있는 지를 검사하는 명령어 'gamma_member'를 추가한다. 이렇게 확장된 EWAM이 있을 경우 앞 절에서 기술된 효율 변환 이후 아래에서 설명하는 "EWAM 변환"을 추가로 수행할 수 있다.

3.2. EWAM 명령어로의 컴파일 방법

비혼 논리 프로그램은 효율 변환을 거친 후 아래에서 설명하는 EWAM 변환을 거치고 그 결과가 EWAM 명령어로 최종 컴파일된다. 다음은 "EWAM 변환"의 규칙이다.

- 클로즈 축약 규칙의 변환: 앞에서 설명한 효율적 변환의 혼 클로즈 "H(…, G1) :- B1(…, G2), …, Bn(…, G2)"는 다음과 같은 혼 클로즈로 다시 변환된다. 여기서 입력 프로그램의 술어 H의 arity는 EWAM으로 변환된 프로그램에서도 동일하다.

```
H(…) :- B1(…),…, Bn(…).
```

- 조상 삭제 공리의 변환: EWAM 변환의 조상 삭제 공리를 구현하기 위한 혼 클로즈는 다음과 같이 표현한다. 여기서 술어 'gamma_member'는 arity가 n인 리터럴 L이 시퀀트 Γ의 member인지를 검사하는 새로운 EWAM 명령어이다.

```
L(A1, …, An) :- gamma_member(L, n).
```

- 재시작 규칙의 변환: EWAM 변환을 위한 재시작 규칙은 다음과 같은 혼 클로즈로 표현한다. 여기서 술어 'gamma_add'는 시퀀트 Γ에 리터럴 L을 추가하는 새로운 EWAM 명령어이다.

```
not L(…) :- gamma_add(0), false.
```

보기 3 (EWAM 변환) 다음은 보기 1의 비혼 논리 프로그램 LP, 목표 클로즈 G 그리고 PCONTRA(LP∪{G})의 EWAM 변환 결과이다. 변환된 프로그램은 입력 비혼 논리 프로그램과 거의 비슷한 모양을 가지고 있다. 변환된 프로그램은 입력 프로그램에 조상 삭제 공리를 위한 클로즈 (a), 재시작 규칙을 위한 클로즈 (r) 및 대우 클로즈 (5-2)가 추가된다.

```
(r) not L(_) :- gamma_add(0), false.
(1) on(a,b).
(2) on(b,c).
(a) color(A1,A2) :- gamma_member(color,2).
(3) color(a,green).
(4) color(b,blue).
(5-1) color(b,blue) :- not(color(b,green)).
(5-2) color(b,green,G) :- not(color(b,blue)).
(6) false :- on(X,Y),
        color(X,green),color(Y,blue).
```

앞의 EWAM 변환을 거친 프로그램은 혼 논리 프로그램과 매우 비슷하여 WAM 명령어로의 컴파일 방법과 유사하다. 하지만 일부 특별한 EWAM 명령어를 생성하여야 하는 부분이 있다. 특별히 처리하여야 하는 부분은 위 보기에서 EWAM 명령어를 사용하여 처리하여야 하는 부분으로 'gamma_add' 부분 및 'gamma_member' 부분이다. 이 부분은 WAM이 아닌 EWAM 명령어를 생성한다.

Ⅳ. EWAM 에뮬레이터 및 컴파일러 구현

4.1. EWAM 에뮬레이터의 구현

EWAM의 레지스터 구조는 혼 논리 프로그램을 효율적으로 구현하기 위한 WAM[1]의 구조에 기존 WAM에 존재하는 인수 레지스터들(argument registers)과 동등한 기능을 하는 레지스터 G를 추가한 것이다. 새로운 레지스터 G는 힙 메모리 내의 Γ의 위치를 가리킨다. 여기서 리스트로 표현된 Γ는 증명 절차 InH-Prolog의 재시작 규칙을 위한 'gamma_add' 명령어를 사용하여 리터럴을 추가하고 조상 삭제 공리를 위한 'gamma_member' 명령어를 사용하여 어떤 리터럴이 Γ의 요소인지를 검사한다.

EWAM은 기존의 혼 논리 프로그램을 위한 WAM 명령에 이외에 비혼 논리 프로그램의 증명 절차 InH-Prolog의 공리 및 재시작 규칙을 위한 2개의 새로운 명령어인 'gamma_member p/n'과 'gamma_add A0'을 추가로 가진다. 새로 추가된 EWAM 명령어의 구현 방법은 다음과 같다. 명령어 'gamma_member p/n'은

arity가 n인 리터럴 'p(A0, ···, An-1)'이 레지스터 G가 가리키는 힙 메모리에 구성된 리스트 Γ의 요소인지 검사한다. 이를 위해 이 명령어는 우선 레지스터 G가 가리키는 리스트의 첫번째 요소가 리터럴 'p'을 위한 셀을 가리키면 뒤에 등장할 리터럴 'p'의 각 인수들을 차례로 A0에서 An-1까지의 인수 레지스터들과 일치화 검사를 한다. 일치화 검사가 실패할 경우 레지스터 G가 가리키는 Γ 리스트의 다음 요소와 일치화 검사를 하는데 이러한 일치화 검사는 리스트의 모든 요소들을 통해 일치화가 이루어질 때까지 계속되며 그 외의 경우는 일치화가 실패한 경우이다. 명령어 'gamma_add A0'은 이미 레지스터 A0가 추가할 리터럴을 위한 'STR' 셀을 가리키고 있으므로 레지스터 A0의 내용을 힙에 복사하고 그 다음 힙 번지에 레지스터 G의 내용을 복사한다. 그리고 마지막으로 레지스터 H(힙 레지스터)의 내용을 레지스터 G에 복사하여 새로운 리터럴을 레지스터 G가 가리키는 리스트에 추가한다. 이 명령어의 사용 후 레지스터 H의 값은 2 증가한다.

4.2. 컴파일러의 구현

비혼 논리 프로그램을 EWAM 명령어로 컴파일하는 컴파일러는 크게 비혼 논리 프로그램을 EWAM 변환을 통하여 변환하여 혼 논리 프로그램과 유사한 프로그램을 생성하는 프로그램과 이 결과를 EWAM 명령어로 바꾸는 프로그램으로 구성된다. 이들 각각은 세부적으로 다음과 같이 구성되어 있다.

- EWAM 변환(SICStus Prolog언어[17]로 구현)
  - 입력 프로그램에서 비혼 클로즈를 찾는다.
  - 비혼 클로즈에서 양의 대우 클로즈를 생성한다.
  - 양의 대우 클로즈로 EWAM 변환을 한다.
- EWAM 명령어 생성(C 언어로 구현)
  - EWAM 명령어를 생성한다.

다음은 보기 3의 (5-1)에서 EWAM 명령어가 생성된 보기이다.

```
predicate('color'/2
    get_constant 'b' 0
    get_constant 'blue' 1
    put_structure 'color'/2 0
    unify_constant 'b'
    unify_constant 'green'
    execute '$not$'/1
```

본 논문에서는 스웨덴의 Upsalla 대학에서 개발한 Luther WAM 에뮬레이터 및 컴파일러 1.1[16]을 확장 수정하여 EWAM 에뮬레이터 및 컴파일러를 구현하였다.

V. 성능 측정

본 논문에서 구현된 EWAM 및 컴파일러의 성능을 측정하기 위하여 3개의 컴파일러를 Sun Ultra 5(Solaris 2.7) 워크스테이션에서 구현하였다. 첫 번째 컴파일러는 단순 변환 후 WAM 컴파일을 하는 컴파일러이고, 두번째 컴파일러는 효율 변환 후 WAM 컴파일을 하는 컴파일러이고, 세 번째는 본 논문에서 제안한 EWAM 변환 후 EWAM 컴파일을 하는 컴파일러이다. 이들 모두의 컴파일된 코드는 모두 EWAM 에뮬레이터에서 수행되었다(참고: WAM은 EWAM,의 부분 집합이므로 모든 프로그램을 같은 EWAM에서 수행시키는 것이 정확한 측정 방법임). 성능 측정에 사용된 프로그램은 관련 자료[5][7]로 부터 얻은 13개의 비혼 논리 프로그램이다. 성능 측정 결과 표1과 같이 단순 변환보다 효율 변환이, 효율적 변환보다 EWAM 변환된 프로그램의 수행 시간이 더 짧은 것을 알 수 있다. 본 성능 측정에서 사용한 프로그램만으로 본 논문에서 개발한 EWAM 에뮬레이터 및 컴파일러의 성능을 정확하게 이야기할 수는 없지만 효율 변환이 단순 변환보다 약 1.8배, EWAM 변환이 효율 변환보다 약 13.4배 빠름을 알 수 있다(단순 변환보다 EWAM 변환이 약 23.5배 빠름).

표 1. 단순, 효율, EWAM 변환의 수행 시간(msec)

시험 프로그램 이름	단순 변환 (si)	효율 변환 (ef)	EWAM 변환 (ew)	속도 증가 (si/ef)	속도 증가 (ef/ew)	속도 증가 (si/ew)
College	6.943	2.628	0.259	2.642	10.147	26.807
Color-blocks	13.184	8.187	1.021	1.61	8.019	12.913
eager-vs-lazy	9.846	5.304	0.349	1.856	15.198	28.212
or1	5.666	3.37	0.238	1.681	14.16	23.807
or2	10.901	6.458	0.418	1.688	15.45	26.079
Path	2.79	2.099	0.2	1.329	10.495	13.95
Subs1	8.332	4.95	0.33	1.683	15	25.248
Subs2	8.335	4.947	0.329	1.685	15.036	25.334
Subs3	8.227	4.931	0.304	1.679	16.22	27.227
Subs4	8.369	4.946	0.332	1.692	14.898	25.208
Subs5	8.327	4.942	0.341	1.685	14.493	24.42
Subs6	18.185	10.806	0.695	1.683	15.548	26.165
Wine	4.514	2.112	0.222	2.137	9.514	20.518
평균	-	-	-	1.773	13.398	23.53

본 측정에서 EWAM 변환이 효율 변환보다 더 효율적으로 수행된 이유는 효율 변환 후 생성된 아래 WAM 코드를 보면 보다 쉽게 알 수 있다.

```
predicate('color'/2
    get_list 0
    unify_constant 'b'
    unify_x_variable 2
    get_list 2
    unify_constant 'blue'
    unify_nil
    get_x_variable 3 1
    put_structure 'color'/2 4
    unify_constant 'b'
    unify_constant 'green'
    put_list 0
    unify_x_value 4
    unify_nil
    put_x_value 3 1
    execute '$not$'/2
)
```

위 코드는 보기 2의 (5-1)에 해당되는 코드로 EWAM 변환 코드에 비하여 코드 길이가 약 3배 정도 길 뿐만 아니라 3번째 인수 G의 일치화를 위하여 상수 일치화 명령어가 아닌 매우 비효율적으로 수행되는 WAM의 변수 일치화 명령어인 'get_x_variable', 'unify_x_value', 'put_x_value'를 사용하기 때문에 매우 비효율적이다. 또한 술어 부름으로 구현되던 'member'가 EWAM의 명령어로 수행되면서 속도의 증가가 이루어졌다. 이러한 사항이 고려되어 약 10배 이상의 효율이 생성된 것으로 판단된다.

VI. 결 론

본 논문은 비혼 논리 프로그램을 효율적으로 수행하는 EWAM을 제안하고 비혼 논리 프로그램을 EWAM 명령어로 컴파일하는 방법을 제시하였다. 본 논문에서 제시된 방법을 사용하여 EWAM 에뮬레이터 및 컴파일러를 구현하여 성능 측정을 한 결과 기존 제안된 WAM 컴파일 방법보다 매우 높은 속도 향상 결과를 얻었다. 본 연구의 결과를 사용하면 기존 혼 논리 언어인 Prolog 같은 언어에서도 비교적 쉽게 비혼 클로즈의 사용을 가능하게 할 수 있다.

참고문헌

[1] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Fourth Edition, Springer-Verlag, Berlin, 1994.

[2] J. W. Lloyd, *Foundations of Logic Programming*, Second, Extended Edition, Springer-Verlag, Berlin, 1987.

[3] D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, October 1983.

[4] H. Aït-Kaci, *Warren's Abstract Machine A Tutorial Reconstruction*, The MIT Press, Cambridge, MA, 1991.

[5] J. Lobo, J. Minker, and A. Rajasekar, *Foundations of Disjunctive Logic Programming*, The MIT Press, Cambridge, MA, 1992.

[6] J. Minker and A. Rajasekar, Procedural Interpretation of Non-Horn Logic Programs, *Proceedings of the 9th International Conference on Automated Deduction*, 278-293, 1988.

[7] J. Minker and A. Rajasekar, A Fixpoint Semantics for Disjunctive Logic Programs, *The Journal of Logic Programming*, Vol. 9, 45-74, Elsevier Science Publishing Co., Inc., New York, 1990.

[8] D. A. Plaisted, Non-Horn Clause Logic Programming Without Contrapositives, Journal of Automated Reasoning, 4:287-325, Kluwer Academic Publishers, 1988.

[9] D. W. Loveland and D. W. Reed, A near-Horn Prolog for Compilation, Technical Report CS-1989-14, Department of Computer Science, Duke University, 1989.

[10] D. W. Loveland, Near-Horn Prolog and Beyond, *Journal of Automated Reasoning*, 7: 1-26, Klwer Academic Publishers, 1991.

[11] D. W. Reed, D. W. Loveland and B. T. Smith, The near-Horn Approach to Disjunctive Logic

Programming, Technical Report CS-1992-19, Department of Computer Science, Duke University, 1992.

[12] D. W. Reed and D. W. Loveland, Near-Horn Prolog and the Ancestry Family of Proof Procedures, *Annals of Mathematics and Artificial Intelligence*, 14, 1995.

[13] 신동하, 비혼 논리 프로그램을 위한 증명 절차, *자연과학연구*, 제5호, 상명대학교 자연과학연구소, 1998.

[14] B. T. Smith and D. W. Loveland, A Simple near-Horn Prolog Interpreter, Technical Report CS-1988-21, Department of Computer Science, Duke University, 1988.

[15] 신동하, 조상계열 증명 절차를 이용한 비혼 논리 프로그램의 컴파일 방법, *정보과학회논문지(B)*, 제 26권, 제 7호, 한국정보과학회, 1999.

[16] J. bevemyr, The luther WAM Emulator, Technical Report 72, Department of Computer Science, Uppsala University, 1992.

[17] SICS, *SICStus Prolog V3 User's Manual*, 1995.

저자소개

신동하(Dongha Shin)

1976년 ~ 1980년 경북대학교 전자계산기공학과 (학사)
1980년 ~ 1982년 서울대학교 전자계산기공학과 (석사)
1994년 U of South Carolina 컴퓨터 과학과 (박사)
1982년~1996년 한국전자통신연구원 책임연구원
1997년~현재 상명대학교 부교수
※ 관심분야: 프로그래밍 언어, 컴파일러, 임베디드 시스템, 계산이론, 인공지능

백윤철(Yun Cheol Baek)

1984년 ~ 1988년 서울대학교 전산과학 (학사)
1988년 ~ 1990년 서울대학교 전산과학 (석사)
1990년 ~ 1995년 서울대학교 전산과학 (박사)
1996년~현재 상명대학교 부교수
2005년 ~ 현재 Princeton University 연구 교수
※ 관심분야: 운영체제, 실시간 시스템, 임베디드 시스템