논 문

논문번호 17-1-6

함수논리 언어를 위한 순차 추상기계의 설계 및 성능평가†

(Design and Performance Evaluation of a Sequential Abstract Machine for Functional Logic Languages)

낭 종 호* 신 동 욱* 맹 승 렬** 조 정 완**
(Jong Ho Nang) (Dong Wook Shin) (Seung Ryoul Maeng) (Jung Wan Cho)

요 약

함수 논리 언어는 함수 언어의 특징과 논리 언어의 특징을 모두 가지고 있는 강력한 프로그래밍 파라다임이지만, 이런 종류의 언어에 대한 효율적인 구현 방법이 아직 개발되지 않았기 때문에 널리 사용되지 못하고 있다 본 논문에서는 이 문제점을 해결하기 위하여 함수 논리 언어를 효율적으로 수행할 수 있는 추상기계 F-WAM에 대한 구조와 인스트럭션 집합을 제안하였다. F-WAM은 논리 언어의 효율적인 처리기인 WAM의 확장형으로서, 기본적인 수행 방법은 SLD-resolution과 리덕션이다 즉, 함수 논리 언어의 논리 언어 부분을 수행할 때는 WAM과 같은 방법으로 수행하고, 함수 응용을 계산할 때는 리덕션 기계과 같은 방법으로 수행한다

본 논문에서는 시뮬레이션을 통하여 F-WAM의 성능을 분석하였는데, 함수 응용을 계산할 때는 많은 메모리 영역을 필요로하는 백트랙킹 정보를 기억할 필요가 없기 때문에 WAM보다 메모리 영역을 적게 사용함을 알 수 있다. 또한 F-WAM의 리덕션 인스트럭션들은 대응되는 WAM 인스트럭션들보다 간단하기 때문에 WAM 보다 빠르게 수행할 수 있다.

ABSTRACT

Though functional logic languages are regarded as powerful programming paradigms, they are not used in many-applications owing to their inefficiencies. To resolve this problem, we propose an abstract machine architecture and the instruction set called F-WAM(Functional-WAM), which executes functional logic languages efficiently. F-WAM is a slightly modified version of WAM(Warren Abstract Machine), and its fundamental execution mechanisms are SLD-resolution and reduction Its execution is similar to WAM in logical deduction, and similar to the reduction machine in function evaluation.

The simulation results of F-WAM show that F-WAM usually uses less memory space than WAM because it does not need to remember backtracking informations in function reduction, and more fast than WAM because reduction instructions of F-WAM are more simple than the corresponding WAM instructions.

†본 연구는 과학기술처의 1988년 특정연구개발사업(계정번호 . N04891)의 연구비 지원에 의한 것임
*준 회 원 . 한국과학기술원 전산학과
**종신회원 ' 한국과학기술원 전산학과 교수
접수일자 1989년 6월 21일

1. 소 개

논리 언어와 함수 언어는 대표적인 선언적(declarative) 언어들로서, 각 언어는 다른 언어가 가지고 있지 않는 특징을 가지고 있다. 즉, 논리 언어는 선언적 의미와 관계(relation)를 위주로 한 언어이기 때문에 지식을 바탕으로 하는 추론 분야에 적합한 반면에, 함수 언어는 여러가지 강력한 프로그래밍 개념(예를들면, high order function, 스트림, 타입 등)을 제공하며,또한 알고리즘을 구현하거나 자료 구조를 처리하는데 논리 언어보다는 적합하다[Bell 86]. 따라서 위에서 언급한 두 언어의 장점을 모두 포함하고 있는 새로운 함수 논리 언어(functional logic language)를 개발하려는 연구들이 많이 진행되고 있다[Barb86, Bosc 86, Darl86, Gogu 86, Levi87, Redd86, Subr86].

함수 논리 언어에 대한 연구는 크게 두 가지로 나눌 수 있는데, 첫번째 부류는 함수 언어에서 사용되는 패턴 매칭(pattern matching)을 단일화(unification)로 확장하는 부류(functional plus logic)이며, 두번째 부류는 논리 언어의 단일화에 같음(equality) 관계를 첨가하여 확장한 같음-단일화(Equality-Unification)를 사용하는 부류(logic plus functional)이다[Bosc 86]. 함수 논리 언어에 대한 내용은 [Bell86]에 자세히 나와 있다

위에서 언급한 두가지 부류의 함수 논리 언어는 다른 언어가 가지고 있지 못하는 여러가지 특징을 가지고 있지만, 아직까지 수행 알고리즘만이 개발되어 있거나 메타 인터프리터로만 구현되어 있어 효율성이 떨어지기 때문에 실제 응용 분야에서 사용하기는 많은 문제점을 가지고 있다 첫번째 부류의 언어 중에서, Reddy는 [Redd 86]에서 SECD 기계에 기초를 둔 추상 narrowing 기계을 개발하여 어느 정도의 효율성을 제공하였다. 하지만 두번째 부류의 언어에서는 Eqlog[Gogu86]와 같이 같음-단일화 알고리즘만이 제안되어 있거나, Funlog[Subr 86]나 Aflog[Shin 87, Shin 88]와 같이 메타 인터프리터로만 구현되어 있어서 실제 응용 분야에 사용하기 어렵다. 이런 문제는 언어의 수행에 필요한 기본 기능을 기계 레벨에서 충분히 제공하지 못하기 때문에 발생하며, 이 문제를 해결하기 위해서는 함수 논리 언어 수행에 필요한 기본 기능을 기계 레벨에서 제공하여 주어야 한다. 즉, 함수 논리 언어의 수행에 필요한 기본 기능인 resolution과 리덕션을 기계 레벨에서 제공하여 주는 추상 기계이 필요하나 아직 이에 대한 연구는 미흡한 실정이다. 최근에 함수 논리 언어의 일종인 K-LEAF[Bell87, Levi87]에 대한 순차 추상기계인 K-WAM[Bosc89]이 발표되었으나, K-LEAF에서는 함수 논리 언어의 함수 언어 부분을 논리 언어로 변환하여 K-WAM에서 수행하기 때문에 기계 레벨에서 진정한 의미의 함수 응용 리덕션 방법은 제공하지 못한다.

본 논문에서는 이러한 문제점을 해결하기 위하여 두번째 부류에 속하는 함수 논리 언어의 하나인 Aflog [Shin87, Shin88] 언어에 대한 추상 기계 F-WAM (Functional-Warren Abstract Machine)의 구조와 인스트럭션 집합을 설계하고, 시뮬레이션을 통하여 성능을 평가 하였다 본 논문에서 F-WAM 설계에 사용한 접근 방법은 다음과 같다. 첫째로는 논리 언어의 효율적인 추상 기계인 WAM(Warren Abstract Machine)에 리덕션 기능을 첨가시켜서 F-WAM을 설계하였다는 것과, 함수 응용을 계산하는 방법으로 "환경에 기초한 리덕션 방법"을 사용하였다는 것이다 이와 같은 접근 방법을 사용한 이유는 논리 언어를 위한 WAM 이라는 강력한 추상 기계이 이미 존재하고 있기 때문이며, 또한 WAM에서는 환경을 이용하여 논리절에 있는변수를 바인딩 하기 때문에 환경에 바탕을 둔 리덕션 방법을 사용하였다. 이러한 이유 때문에 F-WAM에서는 그래프 리덕션(graph reduction) 방법 보다 지연계산(lazy evaluation)을 효율적으로 할 수 없지만, 빠른 리덕션을 할 수 있다는 장점을 가지고 있다.

본 논문은 다음과 같이 구성되어 있다 제 2 장에서는 함수 논리 언어의 일종인 Aflog에 대하여 설명하고, 제 3 장에서는 Aflog에서 사용되는 추론 방법을 제공하기 위한 F-WAM의 기능에 대하여 설명하도록 한다. 제 4 장에서는 제안한 추상 기계 F-WAM에 대한 구조와 인스트럭션 집합에 대하여 설명하고, 제 5 장에서는 F-WAM에 대한 시뮬레이션과 성능 평가에 대하여 설명하도록 한다. 제 6 장에서는 관련된 연구들에 대한 비교를 하며, 마지막으로 7 장에서는 결론을 기술하였다

2. 함수 논리 언어 Aflog와 Canonical Unification

Aflog[Shin 87, Shin 88]는 서론에서 언급한 함수 논리 언어 중에서 두번째 부류에 속하는 언어로서, 같음

-단일화 알고리즘으로 Canonical Unification을 사용한다 본 절에서는 Aflog에 대한 구문과 수행의미(operational semantics), 그리고 Canonical Unification 에 대하여 예제를 통하여 간략하게 알아보도록 한다 이 언어에 대한 내용은 [Shin 87]과 [Shin 88]에 자세히 설명되어 있다.

Aflog 프로그램은 함수에 대한 정의인 등식(equation)과 논리 언어의 일종인 Prolog 논리절로 이루어져 있는데, 함수에 대한 정의인 등식은 다음과 같은 구문 형식을 갖는다 여기서 '==>'는 등호를 나타내는 기호이다

```
A(t1, · , tn)==>(G1 | B1),
                (G2 | B2),
                .. ..
                (Bn).
```

위의 형식에서 t1, …, tn은 데이타 항(data term) 이며, G1…Gn은 부울 수식을 나타내고, B1, · , Bn은 일반적인 항이다. '|'는 가드(guard) 명령으로서 Concurrent Prolog에서 사용하는 가드와 같은 의미를 가지고 있다 .

Canonical Unification은 제한된 같음-단일화 알고리즘으로서, 같음-단일화 알고리즘의 효율성을 위하여 함수 논리 언어에서 사용하는 등식에 다음과 같은 제약을 가하였다. 이 제약은 비록 함수 논리 언어의 표현력을 감소시키지만, Canonical-Unification의 효과적인 수행에 중요한 역할을 한다.

(가) 등식은 canonical하다는 성격을 가져야 한다 등식의 집합 E가, 모든 항 M, N, P에 대하여 P가 M으로 대치되고 ($P \xrightarrow{*} M$) P가 N으로 대치되며 ($P \xrightarrow{*} N$), $M \xrightarrow{*} Q$이고 $N \xrightarrow{*} Q$되는 Q가 존재할 때, 등식의 집합 E는 confluent하다고 하며, 임의의 항 M 에 대해서 M->M1->M2-> 되는 무한 대지 과정이 없을때 등식의 집합 E는 terminating하는 성질을 가졌다고 한다 이때 confluent 성질과 terminating 성질을 만족할 때 등식의 집합은 canonical하다고 한다.

(나) 함수 항은 reduction될 때 논리변수를 가져서는 안된다. 즉 함수의 모든 인수들은 reduction될 때항상 바인딩 상태로 있어야 된다.

주어진 등식이 위와 같은 두가지 조건을 만족하면 함수 응용의 값, 즉 함수 항의 정규형은 오직 하나의 정규형으로 결정된다. 이와 같은 성질을 이용하여 Canonical Unification에서는 다음과 같은 두 단계를 거쳐서 두 항을 단일화 한다. 먼저 두 항을 주어진 등식을 이용하여 더 이상 적용할 등식이 없을 때까지 정규화(normalizing, reduction, rewriting)하여 두항을 모두 완전한 정규형으로 만든다. 다음에는 이 정규형을 가지고 보통의 Prolog에서 사용하는 단일화(즉, 구문적 단일화 : 두 항이 unify될 수 있는가를 조사하는 과정)을 수행한다

알고리즘 1)은 위에서 설명한 Canonical Unification 알고리즘을 나타낸 것이다

알고리즘 1) · Canonical-Unification 알고리즘
입력 : 항 T와 S.
출력 : mgu $\sigma = \{x_1/m_1, x_2/m_2, \cdot, x_P/m_P\}$ 혹은 단일화 할 수 없다는 메시지

1. T와 S를 정규화 시킨다
   Tn=complete-rewriting(T)
   Sn=complete-rewriting(S)
2. Tn과 Sn의 구문적 단일화를 수행한다 Tn과 Sn이 단일화 된다면 mgu $\sigma = \{x_1/m_1, x_2/m_2, \cdots, x_P/m_P\}$를 출력하고 그렇지 않을 경우에는 단일화할 수 없다는 메시지를 출력한다.

Aflog의 Canonical Unification은 완전성(completeness)이 증명되어 있으며, Canonical Unification이 가지고 있는 제약점, 즉 등식이 canonical 하여야 한다는 것과 함수 항의 인수로 바인딩 않된 변수가 나타나서는 않된다는 규칙을 어기지 않고도 여러 알고리즘을 쉽게 프로그래밍할 수 있는 것으로 알려져 있다[Shin 87].

예제 1)과 예제 2)는 Aflog로 작성된 함수 논리 언어 프로그램으로서, 예제 1) 은 두개의 입력 리스트를 받아서 하나의 출력 리스트를 출력하는 프로그램이며, 예제 2)는 리스트를 입력으로 받아서 정렬된 이진 트리를 출력하는 프로그램이다. 이런 예제에서 보는 것과 같이 한 프로그램안에 논리 언어 형태의 프로그램과 함수 언어 형태의 프로그램이 공존할 수 있기 때문에 좀 더 효율적인 프로그램을 작성할 수 있다

예제 1) append프로그램

```
append([ ], Y)=>Y.
append([H|T], Y)=>[H|append(T, Y)].
```

예제 2) make-binary-tree 프로그램

```
/*정렬 이진 나무에 새로운 원소를 첨가 시키는
```

```
    등식*/
insert(A, empty)==>tree(empty, A, empty)
insert(A, tree(L, B, R))==>
            ((A>B) | tree (insert(A, L), B, R)),
            (tree(L, B, insert(A, R)))
  /*논리적인 정의*/
make-binary-tree(L, Tr) : - build-up(L, empty,
  Tr)
  /*주어진 리스트를 가지고 정렬 이진 나무를 만드
    는 논리절*/
build-up([ ], Tr, Tr),
build-up([A|L], Tr, NewTr) : - build-up(L,
        insert(A, Tr), NewTr).
```

3. F-WAM에서의 추론 방법

서론에서 언급한 것과 같이 F-WAM의 추론 방법은 SLD-resolution과 환경에 기초한 리덕션이다. 그런데 환경에 기초한 리덕션 방법에서는 외부-우선(outermost) 리덕션 보다는 내부 우선(innermost) 리덕션을 효율적으로 구현할 수 있기 때문에, 내부-우선 리덕션 방법을 사용하여 함수 응용을 리덕션한다

F-WAM에서의 추론 방법을 좀 더 자세히 알아보면 다음과 같다 F-WAM에서 어떤 논리절의 서브 고울이 인수로 함수 응용 항을 가지고 있는 경우(예를들어 P를 술어(predicate)이름, f를 함수 이름이라 할 때 p(f(a), b)인 경우)에는, 앞에서 언급한 것과 같이 우선 함수 응용 항을 내부-우선 방법으로 정규형을 얻어야 한다 이때 이런 정규화 과정이 성공하여 함수 응용 항에 대한 정규형을 구할 수 있는 경우에는, 이 정규형을 가지고 WAM의 단일화 명령어를 수행하면 된다. 그런데 만약 이런 정규화 과정이 성공하지 못할 경우를 가정하여 보자 이런 경우는 함수 응용 항이 인수로 논리적 변수(logical variable)를 가지고 있거나(예를 들어 p(f(X), a)에서 정규화 과정을 수행할 때 X가 논리적 변수인 경우), 함수 응용의 인수 패턴에 맞는 함수의 정의가 없을 경우인데, 이때는 WAM의 'fail' 동작을 수행하여 앞의 서브 고울을 다시 수행한다. 이 추론과정을 예를들어 설명하면 다음과 같다

〈예제 3〉

```
p(X, Y) : - q(X, Z), r(factorial(Z), Y)
factorial(Z)==>(Z==1) | 1,
            (Z>1) | Z*factorial(Z-1).
```

- 경우 1)

만약 q가 Z를 어떤 정수(여기서는 Z=3이라고 가정하자)로 바인딩 하였다면 r(factorial(Z), Y)에 대한 Canonical Unification 과정에서는, 먼저 factorial(Z)에 대한 정규형을 구한다 즉, 앞에서 가정한 것과 같이 Z를 3이라 하면 factorial(3)의 정규형은 6이 되어, 결과적으로 F-WAM은 r(6, Y)를 WAM과 같은 방법으로 수행하면 된다 만약 r(6, Y) 가 실패하거나 p(X, Y) 이후에 있는 서브 고울이 실패하면, WAM과 같은 'fail' 동작을 수행한다 즉, recent-choice-point가 포인트하고 있는 서브 고울을 다시 수행하게 된다 그런데 Aflog에서의 함수 정의는 canonical 특징을 가지고 있기 때문에 factorial(3)은 6이외에 다른 정규형이 없다. 그러므로 정규화 과정을 다시 수행할 필요가 없다 즉, factorial 함수에 대한 choice-point는 필요 없게 된다 여기서 q(X, Z)가 recent-choice-point가 포인트 하고 있는 서브 고울이라고 가정하면, q(X, Z) 가 다시 수행되어 다른 Z값을 생성하게 된다

- 경우 2)

만약 q(X, Z)가 -1과 같은 값을 Z에 바인딩 하게 되면, 앞에서 언급한 Canonical Unification의 함수 정규화 과정에서 factorial(-1)에 맞는 함수 정의를 찾을 수 없기 때문에 F-WAM은 'fail' 동작을 수행한다

- 경우 3)

만약 q(X, Z)가 Z의 값을 바인딩하지 않으면, 즉, factorial(Z)를 정규화할 때 Z가 논리적 변수로 남아 있는 경우에는 경우 2)와 마찬가지로 F-WAM은 'fail' 동작을 수행한다 그러므로 F-WAM의 패턴 매칭 인스트럭션은 함수의 인수로 넘어오는 값이 변수인지를 항상 검사하여야 한다

4. F-WAM의 수행시 구조 및 인스트럭션 집합

F-WAM은 WAM의 확장형으로서, WAM에 환경에 기초한 리덕션을 할 수 있는 구조 및 인스트럭션을 첨가한 것이다. 본 절에서는 F-WAM의 수행시 데이타 구조와 인스트럭션에 대하여 알아보도록 한다 F-WAM의 수행시 구조는 WAM과 비슷하지만, 패턴 매칭과 함수의 리덕션에 관한 구조와 인스트럭션을 가지고 있는 것이 WAM과는 다른 점이다.

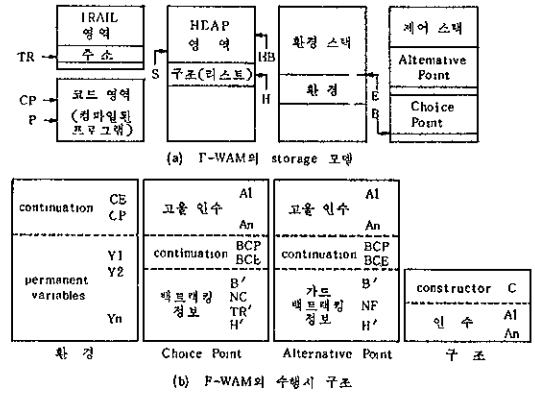
4.1 F-WAM의 수행시 구조

앞에서 언급한 것과 같이 F-WAM의 수행시 구조는 논리 언어를 수행하기 위한 WAM 구조에 함수 언어를 수행하기 위한 구조를 포함하고 있다 (그림 1)에 나타난 것과 같이 F-WAM의 수행시 구조는 WAM의 split-stacking모델[Tick87]의 변형으로서, 원래 WAM에서 STACK 영역에 있던 환경과 choice-point를 분리한 구조에 함수 리덕션을 위한 구조를 첨가한 것이다 메모리 영역은 인스트럭션을 저장하는 코드 영역 (Code Area), 구조(structure)와 리스트를 저장하는 Heap영역, 실행시의 수행 정보를 저장하는 환경 스택(Environment Stack)과 제어 스택(Control Stack), 그리고 단일화를 수행하는 동안 새로이 바인딩되는 변수 등을 저장하는 Trail 영역 등으로 이루어져 있다. WAM 의 레지스터들과 사용되는 용도는 다음과 같다

- P(Program Pointer) : 다음에 수행할 인스트럭션에 대한 포인터.
- CP(Continuation Pointer) . 다음에 수행할 논리절의 인스트럭션에 대한 포인터
- E(Last Environment Pointer) : 가장 최근에 생성된 환경에 대한 포인터
- B(Backtrack Pointer) : 가장 최근의 백트랙킹 정보에 대한 포인터.
- TR(Top of Trail) : Trail 영역의 top에 대한 포인터.
- H(Top of Heap) : Head 영역의 top에 대한포인터
- S(Structure Pointer) : 구조나 리스트에 대한 포인터
- HB(Heap Backtrack Pointer) · Heap 영역의 백트랙 포인터.
- Al, ···, An(Argument Registers) : 인수 레지스터
- X1, · Xn(Temporary Register) : 범용 레지스터.

이때 A1 레지스터는 논리 절에 인수를 전달하는데 사용되고, X1 레지스터는 논리 절의 임시 변수를 저장하는데 주로 사용된다

WAM에서 환경은 논리 절에 있는 변수들을 위한 기억장소와, 그 논리 절이 끝났을때 돌아가야 할 포인트에 대한 정보를 가지고 있는 continuation point로 이루어져 있다. 이런 WAM의 환경은 함수 논리 언어의 함수를 리덕션 할 때도 똑같은 방법으로 사용될 수 있다



(그림 1) F-WAM의 storage 모델과 수행시 구조

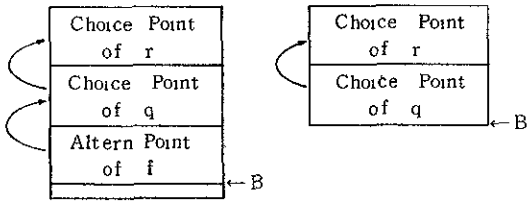
2 장에서 언급한 것과 같이 하나의 등식이 적용되기 위해서는, 호출시의 입력 패턴과 등식의 헤드 패턴이 부합되어야(match)할 뿐만 아니라, 그 등식의 가드 부분도 만족하여야 한다 그런데 가드에는 어떠한 부울 수식이라도 올 수 있기 때문에, 가드를 검사할 때 기계의 상태가 바뀔 수 있다 그러므로 함수 응용을 리덕션하기 위하여 패턴 매칭을 수행하기 전에 기계의 상태를 기억시켜야 한다 이런 목적을 위하여 사용되는 F-WAM의 수행시 구조는 Alternative Point(AP)인데 이 구조는 WAM에서 백트랙킹을 위하여 사용되는 구조인 choice-point와 비슷한 구조를 가지고 있다. 하지만 함수 언어 부분에서는 백트랙킹이 없기 때문에 AP는 한 등식이 적용될 때 제거될 수 있다 또한 함수 언어 부분의 패턴 매칭에서는 출력 바인딩(output binding) 이 없기 때문에 전의 trail 포인트도 기억할 필요가 없다. 예제 4)는 이런 F-WAM의 동작을 설명한 것이다

예제 4)

```
r(X) : -q(X), p(f(X), Y),
r(X).
f(X) ==> (g(X) == 1) | a,
         (g(X) == 2) | b,
         (g(X) == 3) | c.
```

예제 4)에서 f(X)의 가드 부분에 있는 g(X)의 수행도중에 기계의 상태가 바뀔 수도 있기 때문에, 턴 매칭을 수행하기 전에 기계의 상태를 기억시켜야 한다(그림 2-(a)) 이때 f(X)에 기드중의 하나가 성공하

면 f(X)에 대한 AP를 제거할 수 있다(그림 2-(b)). 이런 상태에서 p(f(X)의 정규형, Y)가 실패하면, recent-choice 포인트가 가르키는 q(X)가 X에 대한 새로운 값을 생성하게 된다



(a) f(X)가 fire되기 전의 제어 스택 (b) f(X)가 fire된 후의 제어 스택

(그림 2) 예제 4)에 대한 F-WAM의 상태 변환

4.2 F-WAM의 인스트럭션 집합

〈표 1〉 F-WAM에 대한 인스트럭션 집합

The Complete F-WAM Instruction Set					
WAM Instructions					
Procedure Control		Indexing		Clause Control	
try	l	switch_on_term	Al, v, c, l, s	call	P/arity
retry	L	switch_on_constant	n, ff	execute	
trust	l	switch_on_structure	n, ff	proceed	
try_me_else	L			allocate	
retry_me_else	l			deallocate	
trust_me_else	fail				
Get		Put		Unify	
get_variable	Vi, Al	put_variable	Vi, Al	unify_variable	Vi
get_value	Vi, Al	put_value	Vi, Al	unify_value	Vi
		put_unsafe_value	Yi, Al	unify_unsafe_value	Yi
get_constant	C, Al	put_constant	C, Al	unify_constant	C
get_list	Al	put_list	Al	unify_list	
get_structure	S, Al	put_structure	S, Al	unify_structure	S
get_nil	Al	put_nil	Al	unify_nil	
				unify_void	
Reduction Instructions					
Fget		Rewrite		Fcall	
fget_value	Vi, Al	rewrite_value	Al	fcall	P/arity Bn
fget_constant	C, Al	rewrite_constant	C		
fget_list	Al	rewrite_nil		commit	
fget_structure	S, Al			commit	
fget_nil	Al				

일반적으로 Aflog 프로그램에 있는 하나의 심볼은 F-WAM의 한 인스트럭션과 이들을 묶는 인스트럭션으로 대응된다. 〈표 1〉은 F-WAM의 인스트럭션 집합을 나타낸 것이다. F-WAM의 인스트럭션 집합은 WAM 인스트럭션들과 함수 리덕션을 위한 인스트럭션으로 나눌 수 있는데, 논리 언어 부분을 위한 WAM 인스트럭션 부류로는 Get, Put, Unify, Procedure Control, Indexing, 그리고 Clause Control 인스트럭션이 있다. 이러한 WAM 인스트럭션에 대한 자세한 내용은 [Gabr 85]에 설명되어 있기 때문에 여기서는 생략하고, 함수 언어 부분을 위한 리덕션 인스트럭션들만을 설명하기로 한다.

F-WAM에서 함수 리덕션을 위한 인스트럭션 부류로는 Fget, Rewrite, Fcall, 그리고 Commit 인스트럭션 등이 있다

- fget 인스트럭션 부류 :

이 인스트럭션 부류는 호출할 당시의 인수 패턴이 등식의 헤드 부분 패턴과 일치하는가를 검사하는 인스트럭션으로서, WAM의 Get 인스트럭션과는 달리 인수가 변수인가도 검사하여야 한다 만약 인수가 변수면 'fail' 동작을 수행하여 앞의 서브 고울에서 다른 인수를 생성하게 한다

- rewrite 인스트럭션 부류 :

이 인스럭션 부류는 인수트럭션의 인수가 포인트하고 있는 함수 응용의 정규형을 리턴하는 인스트럭션이다 예를들어 'rewrite A1' 이라는 인스트럭션은 A1 에 저장되어 있거나 A1이 포인트하는 구조를 함수 응용의 정규형으로 리턴하는 인스트럭션이다.

- fcall 인스트럭션 .

Fcall 인스트럭션은 컴파일된 함수 정의(등식)를 호출하는 인스트럭션으로서, 'fcall Proc / Arity Bn' 의 형식을 갖는데, 여기서 Bn은 정규형을 포인트 하는데 사용되는 레지스터나 변수이다. 그러므로 위의 rewrite 인스트럭션들은 리턴되기 전에 An 값을 Bn에 복사하도록 하여야 한다 중첩된 함수 응용인 경우에는 기정 안쪽에 있는 함수 호출을 먼저하고(내부-우선 리덕션), 그 결과를 레지스터나 변수를 통하여 다음 호출에 사용하게 한다

- commit 인스트럭션

Commit 인스트럭션은 함수 언어의 패턴 매칭을 위한 인스트럭션이다 4.1절에서 언급한 것과 같이 AP는 함수 언어 부분의 패턴 매칭을 수행할 때 생성되는 데이타 구조로서 한 등식의 패턴 매칭과 가드가 성공하면 제거할 수 있는 데이타 구조이다. 이런 작업을 하여 주는 인스트럭션이 commit 인스트럭션이다 그러므로 이 인스트럭션은 패턴 매칭에 대한 인스트럭션과 가드를 검사하는 인스트럭션 다음에 나오는 인스트럭션이다

Aflog 프로그램으로 부터 F-WAM 코드가 어떻게 생성되는 가를 다음 예제를 통하여 알아보자. (그림 3)은 앞에서 설명한 Aflog append 프로그램(그림 3-a)과 컴

파일된 F-WAM코드 (그림 3-b)를 나타낸 것이다. 여기서 'Ai'는 F-WAM의 레지스터를 나타낸 것인데, F-WAM에서 레지스터는 인수 전달과 중간 결과를 저장하는데 사용된다. (그림 3-b)에서 옆에 나와 있는 Aflog 심볼들은 그 인스트럭션이 어떻게 추출되었는가를 나타낸 것이다.

```
F1: append([ ],X) ==> X.
F2: append([X|U], V) ==> [X|append(U, V)].
```

(a) Aflog 로 작성된 append 프로그램

```
append : switch_on_term  A1, fail, F1, F2, fail
F1: fget_nil        A1          %  append([ ], X)
    rewrite_value   A2          %                ==> X
    proceed                     %
F2: allocate
    fget_list       A1          % append([
    unify_value     'X'         %          X
    unify_value     'U'         %           |U],
    fget_value      'V', A2     %              V)
    put_value       'U', A1     % append(U,
    put_value       'V', A2     %          V
    fcall           append/2, A2 %        )
    put_list        A3          %==> ( [
    unify_value     'X'         %          X
    unify_value     A2          %           |append(U, V)]
    rewrite_value   A3          %      )
    deallocate
    proceed                     %
```

(b) (a)의 append 프로그램에 대한 컴파일된 F-WAM 코드

(그림 3) Aflog로 작성된 append 프로그램과 컴파일된 F-WAM 코드

(그림 3)에 나와 있는 F-WAM 코드는 되돌림(recursive) 특징을 가지고 있지만, WAM과 같이 tail-recursion 특징을 가지고 있지는 못하다. 그 이유는 모든 함수가 계산이 되어야지만 그 함수에 대한 환경을 제거할 수 있기 때문이다 append를 정의한 두 등식은 첫번째 인수에 의하여 구별할 수 있기 때문에 AP 구조는 필요없게 된다. 이 코드에 대한 또 다른 특징으로는 F2 에서 첫번째 인수의 헤드 부분을 환경에 저장하는것인데, 이 값은 테일 부분의 함수 응용이 리턴될 때까지 저장하게 된다. put-list 인스트럭션은 이 두 값을 연결하는데 사용되는데, 이 리스트가 append 함수 응용에 대한 정규형으로서 rewrite 인스트럭션과 proceed 인스트럭션에 의하여 호출한 프로시쥬어로 리턴하게 된다.

두번째 예제는 앞에서 설명한 Aflog로 작성된 make-binary-tree 프로그램 중에서 insert 함수에 대한 Aflog 프로그램과 그 컴파일된 F-WAM 코드이다.

```
F1: insert(A, empty) ==> tree(empty, A, empty).
F2: insert(A, tree(L, B, R)) ==>
          (A<B)|(tree(insert(A, L), B, R)),
F3:       (A>=B)|(tree(L, B, insert(A, R))).
```

(a) Aflog 로 작성된 insert 함수

```
insert: switch_on_term  A2, fail, F1, fail, F2

F1: fget_value_XA   'A', A1      % insert(A,
    fget_constant   'empty', A2  %        empty)
    put_structure   'tree', A2   % tree(
    unify_constant  'empty'      %      empty,
    unify_value     A1           %      A,
    unify_constant  'empty'      %      empty
    rewrite_value   A2           %    )
    proceed                      %    .

F2: try_me_else     F3
    allocate
    fget_value_XA   'A', A1      % insert(A,
    fget_structure  'tree', A2   %      tree(
    unify_value     'L'          %           L,
    unify_value     'B'          %           B,
    unify_value     'R'          %           R))
    put_value       'A', A1      % (A<
    put_value       'B', A2      %  B
    lt A1, A2                    % )
    commit                       % |
    put_value       'A', A1      % insert(A,
    put_value       'L', A2      %        L
    fcall           insert/2, A1 %      )
    put_structure   'tree', A2   % tree(
    unify_value     A1           %      insert(A, L),
    unify_value     'B'          %      B,
    unify_value     'R'          %      R
    rewrite_value   A2           %    )
    deallocate
    proceed                      %

F3: trust_me_else_fail
    allocate
    fget_value_XA   'A', A1      % insert(A,
    fget_structure  'tree', A2   %      tree(
    unify_value     'L'          %           L,
```

```
unify_value     'B'          %          B,
unify_value     'R'          %          R))
put_value       'A', A1      % (A>==
put_value       'B',A2       % B
ge A1, A2                    % ) ?
put_value       'A', A1      % insert(A,
put_value       'R', A2      %        R
fcall           insert/2, A1 %         ).
put_structure   'tree', A2   % tree(
unify_value     'L'          %      L,
unify_value     'B'          %      B,
unify_value     A1           %      insert(A, R)
rewrite_value   A2           %      )
deallocate
proceed                      %      .
```

(b) (a)의 insert 함수에 대한 컴파일된 F-WAM 코드

(그림 4) Aflog로 작성된 insert 함수와 컴파일된 F-WAM 코드

(그림 4)에 있는 insert 함수는 (그림 3)에 있는 append 프로그램과는 달리 인수만으로 적용할 등식을 구별할 수 없기 때문에 AP구조가 필요하며, 이 AP는 try-me-else 인스트럭션에 의하여 생성되고, 패턴 매칭 인스트럭션과 가드를 검사하는 인스트럭션 다음에 있는 commit 인스트럭션이나 trust-me-else 인스트럭션에 의하여 제거된다

5. 시뮬레이션과 결과 분석

본 논문에서 제안한 F-WAM에 대한 성능을 평가하기 위하여 F-WAM에 대한 시뮬레이터를 C 언어를 이용하여 SUN3/160에서 작성하였다 본 장에서는 F-WAM에 대한 시뮬레이션과 그 분석에 대하여 알아 보도록 한다

5.1 성능 평가 기준과 가정

F-WAM의 성능은 여러 벤치마크 프로그램을 수행하는데 걸리는 시간과 필요한 메모리 양을 기준으로 측정하였으며, 그 결과를 같은 의미를 갖는 Prolog 프로그램을 수행한 WAM의 결과와 비교하였다 예상되는 수행 속도는 컴파일된 프로그램을 F-WAM에서 수행하는데 필요한 메모리 참조횟수와 레지스터 전송횟수를 기준으로 하였는데, 그 속도 비율은 3 : 1로 가정하였다 즉, 메모리를 참조하는데 걸리는 시간은 레지스터 사이 전송 시간의 3배라고 가정하였다 수행에 필요한 메모리 양은 사용된 Heap 영역과 환경, Choice Point, AP, 그리고 Trail 영역의 합으로 계산하였다

5.2 시뮬레이션과 결과 분석

첫번째 벤치마크 프로그램은 앞의 예제에서 언급한 append 프로그램이다 Aflog로 작성된 append 프로그램과 Prolog로 작성된 append 프로그램을 각각 컴파일시켜서 F-WAM과 WAM에서 수행시켰을때, 예상되는 수행시간과 필요한 메모리 양을 〈표 2〉에 나타내었다

〈표 2〉 append 프로그램에 대한 WAM과 F-WAM의 성능

(a) append를 수행할 때 F-WAM의 성능

수행시 성능 / 리스트의 갯수	메모리 사용량				수행속도에 관련된 요인		
	환경 스택 사용량	제어 스택 사용량	Heap 영역 사용량	Trail 영역 사용량	메모리 참조 횟수	레지스터 전송 횟수	레지스터 비교 횟수
20	101	0	138	0	967	385	380
40	201	0	258	0	1887	745	740
60	301	0	378	0	2807	1105	1100
80	401	0	498	0	3727	1465	1460
100	501	0	618	0	4647	1825	1820

(b) append를 수행할 때 WAM의 성능

수행시 성능 / 리스트의 갯수	메모리 사용량				수행속도에 관련된 요인		
	환경 스택 사용량	제어 스택 사용량	Heap 영역 사용량	Trail 영역 사용량	메모리 참조 횟수	레지스터 전송 횟수	레지스터 비교 횟수
20	6	0	138	1	957	472	448
40	6	0	258	1	1857	912	868
60	6	0	378	1	2757	1352	1288
80	6	0	498	1	3657	1792	1708
100	6	0	618	1	4557	2232	2128

(c) WAM과 F-WAM의 비교

- 총메모리 사용량=환경스택사용량+제어스택사용량+Heap영역사용량+Trail영역 사용량
- 예상수행속도=(총메모리참조횟수×3)+총 레지스터 동작횟수

수행시 성 능 / 리스트의 갯수	F-WAM 총 메모리 사용량	F-WAM 예상 수행 시간	WAM 총 메모리 사용량	WAM 예상 수행 시간
20	239	3666	145	3791
40	459	7146	265	7351
60	679	10626	385	10911
80	899	14106	505	14471
100	1119	17586	625	18031

〈표 2〉에서 볼 수 있는 것과 같이 append 프로그램을 수행하는데 필요한 수행시간은 F-WAM이나 WAM경우 거의 비슷하다. 하지만 F-WAM의 경우 더 많은 메모리 영역을 사용함을 알 수 있다 그 이유는 WAM의 경우에는 마지막 고울을 수행하기 전에 환경을 회수하는 tail-recurison-optimization을 적용할 수 있지만, F-WAM의 경우에는 마지막 함수 호출의 결과를 받아서 리턴해야 하기 때문에 이 방법을 사용할 수 없다 따라서 같은 프로그램을 수행하는데 F-WAM이 더 많은 메모리 영역을 필요로 한다 이와같은 append 프로그램은 F-WAM의 단점을 보여주는 것이다 하지만 이런 단점은 다음 벤치마크 프로그램을 통하여 알 수 있는 것과 같이 F-WAM의 다른 장점으로 인하여 상쇄될 수 있다

다음 벤치마크 프로그램은 make-binary-tree 프로그램이며, F-WAM이 WAM보다 효율적임을 보여주는 프로그램이다 앞에서와 마찬가지로 이 프로그램을 Aflog로 작성한 경우와 Prolog로 작성한 경우를 각각 F-WAM과 WAM으로 컴파일하여 수행시켰을 때의 각각의 성능을 〈표 3〉에 나타내었다

〈표 3〉 make-binary-tree 프로그램에 대한 WAM과 F-WAM의 성능

(a) make-binary-tree를 수행할 때 F-WAM의 성능

수행시 성 능 / 리스트의 갯수	메모리 사용량: 환경 스택 사용량	제어 스택 사용량	Heap 영역 사용량	Trail 영역 사용량	수행속도에 관련된 요인: 메모리 참조 횟수	레지스터 전송 횟수	레지스터 비교 횟수
20	108	300	580	112	14241	5913	6974
40	208	600	1960	422	52241	21593	25724
60	308	900	4140	932	114041	47073	56274
80	408	1200	7120	1647	199641	82353	98624
100	508	1500	10900	2552	309041	127433	152774

(b) make-binary-tree를 수행할 때 WAM의 성능

수행시 성 능 / 리스트의 갯수	메모리 사용량: 환경 스택 사용량	제어 스택 사용량	Heap 영역 사용량	Trail 영역 사용량	수행속도에 관련된 요인: 메모리 참조 횟수	레지스터 전송 횟수	레지스터 비교 횟수
20	68	9	540	1	11537	4103	4125
40	128	9	1880	1	43627	15373	15625
60	188	9	4020	1	96317	33843	34525
80	248	9	6960	1	169607	59513	60825
100	308	9	10700	1	263497	92383	94525

(c) WAM과 F-WAM의 비교

- 총메모리사용량 = 환경스택사용량 + 제어스택사용량 + Heap영역사용량 + Trail 영역 사용량
- 예상수행속도 = (총 메모리 참조횟수 × 3) + 총 레지스터 동작 횟수

수행시 성 능 / 리스트의 갯수	F-WAM 총 메모리 사용량	F-WAM 예상 수행 시간	WAM 총 메모리 사용량	WAM 예상 수행 시간
20	618	42839	1100	55610
40	2018	161879	3190	204040
60	4218	357319	6280	445470
80	7218	629159	10370	779900
100	11018	977399	15460	1207330

위의 make-binary-tree 경우의 시뮬레이션 결과를 통하여 F-WAM이 WAM의 경우 보다 메모리를 더 적게 사용함을 알 수 있으며, F-WAM이 WAM보다 빠르게 수행할 수 있음을 알 수 있다 메모리를 적게 사용하는 이유는 앞에서 언급한 것과 같이 F-WAM의 AP 구조는 대응되는 WAM의 *Choice-Point*와는 달리 한등식이 fire 되면 계속 메모리 영역에 남아있을 필요가 없기 때문이다 즉, F-WAM이 비록 tail-recursion-optimization을 제공하지 못하지만 상대적으로 많은 메모리 영역을 필요로하는 AP를 빠르게 제거할 수 있기 때문에 choice-point를 계속 기억하는 WAM보다 적은 메모리 영역을 필요로 한다 또한 F-WAM이 WAM 보다 빠른 이유는 F-WAM의 리덕션 인스트럭션이 대응되는 WAM의 인스트럭션 보다 간단하기 때문이다 특히, F-WAM의 패턴 매칭 인스트럭션은 WAM의 단일화 인스트럭션보다 간단하기 때문에 빠르게 수행할 수 있다

6 관련된 연구들과의 비교

*Prolog*나 LISP과 같이 하나의 기호 처리용(symbolic

procerssing) 언어를 위한 기계에 관한 연구는 많이 진행되었으나, 두 언어를 모두 수행시킬 수 있는 기계에 대한 연구는 비교적 적은 상태이다. 논리 언어와 함수 언어를 모두 수행시킬 수 있는 기계에 관한 연구는 Xenologic사의 X-1[Dobr87]과 CSELT연구소의 K-WAM[Bosc 89] 등이 있다. 이들과 F-WAM을 비교하면 다음과 같다.

Xenologic사의 X-1은 기호 처리용 언어(특히 Prolog와 LISP)를 위하여 특별히 설계된 기계이다. 이 기계의 구조와 인스트럭션 집합은 WAM과 거의 비슷하며, LISP 프로그램을 WAM 코드로 컴파일하여 수행하는 방법을 사용한다. 그러므로 X-1에서는 함수 언어를 리덕션 방법으로 수행하는 것이 아니라 WAM의 resolution 방법을 이용하여 수행한다. F-WAM과 X-1 과의 차이점으로는 F-WAM에서는 리덕션과 resolution을 위한 구조와 인스트럭션이 모두 있는 반면에, X-1에서는 resolution을 위한 구조와 인스트럭션만이 있다. 또한 X-1은 함수 논리 언어를 위한 기계이 아니라, Prolog와 LISP을 독립적으로 수행하기 위한 기계이다. 그렇기 때문에 X-1은 함수 언어와 논리 언어를 각각 독립적으로 수행할 수 있지만, F-WAM과 같이 한 프로그램 안에서 두 가지 형태의 언어를 동시에 수행할 수 없다는 것이 차이점이다.

CSELT 연구소에서는 K-LEAF라는 함수 논리 언어를 flattening 방법을 사용하여 논리언어 형태로 바꾸고 "외부-우선 SLD-resolution"을 사용하여 이 논리언어 프로그램을 수행한다[Levi87]. 최근에 이런 외부-우선 *SLD-resolution*을 위한 순차 추상 기계인 K-WAM이 발표되었는데, 이 기계은 WAM의 구조와 인스트럭션 집합을 외부-우선 SLD-resolution을 제공할 수 있도록 수정한 기계이다. K-WAM에서는 K-LEAF 프로그램에 있는 함수 응용을 closure 형태로 저장하고 있다가, 이 함수 응용의 값이 필요하게 되면 WAM의 resolution 방법을 사용하여 값을 구한다. 즉, 요구가 있을때 함수 응용을 리덕션하는 지연 계산 방법을 사용한다. 또한, 함수의 값을 구할 때도 F-WAM과 같이 내부-우선 방법으로 구하는 것이 아니라 함수 응용의 Head Normal Form(HNF)[Peyt87]까지만(resolution을 이용하여) 외부-우선 리덕션을 수행하기 때문에 무한 자료 구조를 제공할 수 있다. 그러나 이러한 함수 응용을 closure 형태로 저장하기 위한 오버헤드와 외부-우선 방식의 리덕션을 수행하기 위한 오버헤드 때문에 K-WAM의 수행 속도는 같은 기능을 갖는 Prolog 프로그램을 수행하는 WAM의 성능보다 2-3배 정도 느리다[Bosc89].

위에서 언급한 K-WAM과 F-WAM의 가장 큰 차이점으로는 K-WAM에서는 외부-우선 resolution을 이용하여 함수 응용의 HNF까지만 리덕션을 수행하기 때문에 함수 언어에서 무한 자료 구조를 제공할 수 있는 반면에, F-WAM에서는 리덕션 인스트럭션을 사용하여 내부-우선 리덕션을 수행하기 때문에 무한 자료 구조를 제공할 수 없다는 것이다. 하지만 K-WAM에서는 외부-우선 리덕션을 하기 위한 오버헤드 때문에 수행 속도가 WAM보다 2-3배 정도 느린 반면에, F-WAM의 수행 속도는 이런 오버헤드가 없기 때문에 수행 속도가 WAM과 거의 비슷하다.

〈표 4〉는 X-1과 K-WAM, 그리고 F-WAM에 대한 전체적인 비교를 나타낸 것이다. 여기서 예상 수행 시간은 WAM의 수행 시간에 대한 비율인데, 이 비율은 비록 같은 벤치마크 프로그램을 실제로 수행시켜서 얻은 결과가 아니지만 지금까지 발표된 결과 바탕으로 정리한 것이기 때문에 각 추상 기계에 대한 개략적인 성능을 알 수 있다.

〈표 4〉 함수 논리 언어를 위한 추상 기계들에 대한 비교

추상머신 특징	X-1	K-WAM	F-WAM
대상 언어	Prolog와 LISP을 개별적으로 지원	K-LEAF	Aflog
함수 언어의 수행 방법	논리 언어 형태로 변환하여 수행	논리 언어 형태로 변환하여 수행	리덕션 인스트럭션을 이용하여 직접 수행
함수 응용의 리덕션 방법	내부-우선 리덕션	외부-우선 리덕션	내부-우선 리덕션
무한 자료 구조의 지원 여부	지원 못함	지원 함	지원 못함
WAM과 비교한 예상 수행 시간	1	1-3	0.8-1

7. 결론 및 앞으로의 연구 방향

함수 논리 언어는 논리 언어의 특징과 함수 언어의 특징을 모두 가지고 있는 매우 강력한 언어이다. 하지만 이런 언어를 효율적으로 수행할 수 있는 방법이 아직 개발되지 않았기 때문에 널리 사용되지 못하고 있다. 이 문제점을 해결하기 위하여 본 논문에서는 함수 논리 언어의 수행에 필요한 기본 기능을 기계 레벨에서 제공

하는 추상 기계을 설계하고, 그 인스트럭션 집합을 제안하였다

제안된 추상 기계 F-WAM은 Canonical Unification에 바탕을 둔 함수 논리 언어(특히 Aflog)를 효율적으로 수행할 수 있도록 설계되었는데, 설계 방법은 논리 언어 기계인 WAM에 리덕션 방법을 첨가하였다 F-WAM은 같은 기능을 갖는 논리 언어 프로그램을 수행시키는 WAM보다 함수 논리 언어 프로그램을 효율적으로 처리할 수 있음을 시뮬레이션을 통하여 보였다

하지만 현재의 F-WAM은 함수 언어의 가장 큰 특징 중의 하나인 지연 계산을 할 수 없기 때문에 무한 자료구조를 제공하지 못한다는 단점을 가지고 있다 그러므로 제안된 F-WAM에 지연 계산을 할 수 있는 방법에 관한 연구가 더 있어야 하겠다.

참 고 문 헌

[Barb86] R. Barbuti, M Bellia, G Levi, and Martelli, "LEAF: A Language which integrates Logic, Equations and Functions," *LOGIC PROGRAMMING Functions, Relations, and Equations,* D DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986, pp. 201-238.

[Bell86] M. Bellia and G Levi, "The Relation between Logic and Functional Languages: A Survey," *Journal of Logic Programming 3,* 1986, pp 217-236.

[Bell87] M Bellia, P. G. Bosco, E. Giovannetti, G Levi, C Moiso, and C Palamidessi, "A two level approach to logic plus functional programming integration," *Proc. 87 PARLE Conference, Springer-Verlag,* 1987, pp.374-393.

[Bosc86] P G. Bosco and E Giovannetti, "IDEAL. An Ideal Deductive Applicative Language," *Proc 1986 Symposium on Logic programming,* IEEE Computer Society Press, 1986, pp.89-94.

[Bosc89] P. G. Bosco, C Cecchi, C. Moiso, "An Extension of WAM for K-LEAF: A WAM-based Compilation of Conditional Narrowing," *Proc of 6th International Conference on Logic Programming,* MIT Press, 1989, pp.318-333

[Darl86] J. Darlington, A. J. Field, and H. Pull, "The Unification of Functional and Logic Language," *LOGIC PROGRAMMING Functions, Relations, and Equations,* D. DeGroot and G Lindstorm (eds) Prentice-Hall, 1986, pp.37-70.

[Dobr87] T. P. Dobry, "A Coprocessor for AI. LISP, Prolog and Data Bases," *Spring Compcon '87,* IEEE Computer Society, 1987, pp.396-402.

[Gabr85] J. Gabriel, T. G. Lindholm, E. L. Lusk, and R. A. Overbeek, "A Turotial on the Warren Abstract Machine for Computational Logic," *Research Paper ANL-84-84,* Argonne National Laboratory, Jun., 1985.

[Gogu86] J. A Goguen and J. Meseguer, "EQLOG Equality, Types, and Generic Modules for Logic Programming," *LOGIC PROGRAMMING Functions, Relations, and Equations,* D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986, pp.295-363.

[Levi87] G. Levi and P G. Bosco, "A Complete Semantic Characterization of K-LEAF, A Logic Language with Partial Functions," *Proc 1987 Symposium on Logic Programming,* IEEE Computer Society Press, 1987, pp 318-327

[Peyt87] S. L. Peyton Jones, *The Implementation of Functional Programming Language,* Prentice-Hall, 1987, pp.193-206

[Redd86] U. S. Reddy, "Logic Language Based on Functions Semantics and Implementation," *Ph D Dissertation,* Univ. of Utah, Aug , 1986.

[Shin87] D. W Shin, J. H. Nang, S. Han, and S. R. Maeng, "A Functional Logic Language Based on Canonical Unification," *Proc 1987 Symposium on Logic Programming,* IEEE Computer Society Press, 1987, pp 328-334.

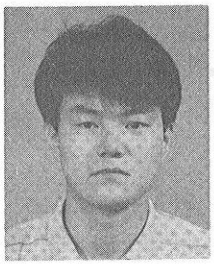
[Shin88] D. W Shin, J H. Nang, S. R. Maeng, and J. W. Cho, "The Semantics of a Functional Logic Language with Input Mode," *Proc of the International Conference On the Fifth Generation Computer Systems 1988,* 1988, pp. 327-336.

[Subr86] P. A. Subrahmanyam and J. H. You, "FUNLOG: A Computational Model Integrating Logic Programming and Functional

Programming," *LOGIC PROGRAMMING Functions, Relations, and Equations,* D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986, pp.157-198.

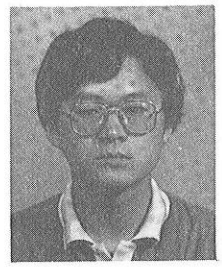
[Tick87] E. Tick, "Studies in Prolog Architectures," Stanford Univ., *TR CSL-TR-87-329,* Jun., 1987.

[Turn 85] D. A. Turner, "Miranda: A Non-strict Functional Language with Polymorphic Types." *Proc. of the IFIP International Conference on Functional Programming Language and Computer Architecture,* Springer Lecture Notes in CS LNCS201, 1985, pp.1-16.



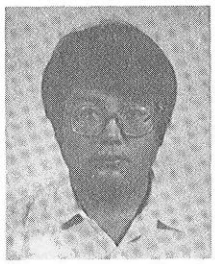
낭 종 호

1986년 서강대학교 전자계산학과 졸업
1988년 한국과학기술원 전산학과 석사학위 취득
1988년~현재 한국과학기술원 전산학과 박사과정 재학중
주 관심분야 : Functional Logic Language, AI Machine, Multiprocessor Architecture, Neural Network



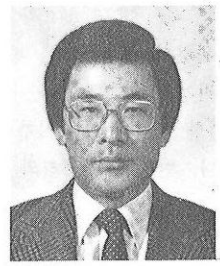
신 동 욱

1984년 서울대학교 전자계산기공학과 졸업
1986년 한국과학기술원 전산학과 석사학위 취득
1986년~현재 한국과학기술원 전산학과 박사과정 재학중
주 관심분야 : Functional Logic Language, Logic Programming Language, Language Semantics, Concurrent System



맹 승 렬

1977년 서울대학교 전자공학과 졸업
1979년 한국과학기술원 전산학과 석사학위 취득
1984년 한국과학기술원 전산학과 박사학위 취득
1984년~현재 한국과학기술원 조교수
주 관심분야 : Parallel Processing, Data Flow Machine, Inference Machine, Vision Architecture, Object-Oriented Language Architecture



조 정 완

1964년 서울대학교 전자공학과 졸업.
1968년 와이오밍 주립대학 전기공학 석사학위 취득.
1973년 노스웨스턴 대학 전산학과 박사학위 취득.
1964년~1973년 동아방송 Ampex, Information Internation Inc., NRRI, IBM 등에서 근무.
1973년~현재 한국과학기술원 전산학과 교수
주 관심분야 : Computer Architecture, Microprogramming, Bitslice Application, Parallel Processing, Inference Machine.