데이터플로우 그래프 표현 방식을 이용한 함수 논리 언어의 실행

김 용 준† · 전 서 현††

요 약

본 논문에서는 함수 논리 언어를 수행할 수 있는 데이타 플로우 표현 방법을 제시하고 함수 논리 언어의 각 절과 함수를 데이타 플로우 그래프로 변환하는 방법에 대해 설명한다. 실행의 효율성을 높이기 위해 AND-병렬성을 위한 종속성 분석을 서브고울과 병행 수행하여 지연시간을 줄였으며, 함수 부분의 수행을 위해 병렬 감축을 사용하였다. RAP을 도입함으로써 발생하는 지연시간을 줄이기 위해 종속성 분석과 서브 고울을 병렬로 처리함으로써 CGE+ 표현 방식에 비해 효율적임을 보였다. 지능형 퇴각 검색을 도입하여 병렬 컴퓨터에서 효율적인 수행을 할 수 있도록 하였다.

Execution of a Functional Logic Language Using the Dataflow Graph Representation

Kim, Yong Jun† · Cheon, Suh Hyun††

ABSTRACT

In this paper, We describe a dataflow model for efficient execution of a functional logic language and a method of translation a functional logic language into a dataflow graph. To explore parallelism and intelligent backtracking, we us model in which clause and function are represented as independent dataflow graph. The node denotes basic actions to be performed when the clause and function are executed. The dataflow mechanism allows an operation to be executed as soon as all its operands are available. Since the operations can never be executed earlier, a dataflow model is an excellent base for increasing execution speed. We did decrease a delay time with concurrent execution of dependency analysis and subgoal.

1. 서 론

선언적 프로그래밍 언어는 표현력이 뛰어나고 명백한 수학적인 의미를 가지므로 소프트웨어를 디자인하고 유지하는데 있어서 정형화된 방법을 제공한다[3]. 수학적 의미에 기초한 참조 투명성은 병렬 처리를 쉽게 할 수 있도록 하며 간결하고 명료한 프로그램은 소프트웨어의 생산성을 높이는데 이용될 수 있으므로 하드웨어가 발전함에 따라, 이에 대한 많은 연구가 있었다.

선언적 프로그래밍 언어는 크게 함수 언어와 논리 언어로 분류할 수 있다. λ-calculus, equational logic, rewriting등의 잘 알려진 수학적인 기초를 가지고 있는 함수 언어는 고차원 함수, 무한 자료 구조, 타입 등의 프로그래밍 파라다임을 제공하므로 알고리즘을 구현하거나 자료구조를 표현하는데 적합하다[6]. 혼절, SLD-

† 정 회 원 : 김천대학 사무자동화과 조교수
†† 정 회 원 : 동국대학교 컴퓨터과학과 교수
논문접수 : 1997년 1월 28일, 심사완료 : 1998년 6월 30일

분해등을 기초로 하는 논리 언어는 논리 변수, 단일화, 연역적 추론등의 컴퓨팅 파워와 우수한 표현력을 제공하고, 지식을 기반으로 하는 추론 작업에 적합하다. 따라서 이들 두 언어의 장점을 모두 가지는 언어를 디자인하기 위한 연구가 있었고, 그 결과로 만들어진 언어를 함수 논리 언어라 한다.

현재 개발된 함수 논리 언어는 아직 선언적 언어의 의미를 명백히 하지 못하고 있으며, 적합한 응용 분야의 개발도 미진한 상태이다. 실행 방법에 있어서는 두 언어의 장점을 모두 수용하기 위해 매우 복잡한 추론 메카니즘을 사용함으로써 단일 언어의 파라다임하에서 실행하는 경우에 비해 효율성이 떨어지고, 이를 극복하기위해 개발된 모델들은 함수 논리 언어가 가져야할 특성들 중에서 일부를 포기함으로써 효율성을 얻었기 때문에 표현상에 제약을 받는다. 따라서 함수 논리 언어가 가져야할 특성들을 모두 유지하면서 프로그램에 내재된 병렬성을 추출하여 비효율적인 수행 방법을 개선하기 위한 노력이 있었다[3][8].

이 논문에서는 함수 논리 언어의 병렬성을 하나의 파라다임 하에서 추출하여 실행하는 데이타 플로우 모델을 제시하고, 함수 논리 언어를 데이타 플로우 그래프로 변환하는 방법에 대해 설명한다. 데이타 플로우 그래프의 효율적인 실행을 위해 함수 논리 언어가 가지는 AND-병렬성과 병렬 감축 기법이 사용된 실행 메카니즘을 설명한다. AND-병렬성의 구현을 위해 종속성 분석을 이용하는 RAP(Restrict AND Parallelism)을 채택하였고 인수가 준비된 함수는 항상 즉각적인 실행이 가능하다.

2. 함수 논리 언어

바람직한 함수 논리 언어는 하나의 구조(framework) 하에서 함수언어와 논리언어를 자연스럽게 표현할 수 있어야하고 각각의 언어를 구현하는 구조에 비해 전체 성능이 떨어지지 않아야 한다[3]. 또한, 함수 논리 언어는 함수언어와 논리언어를 결합하는 방법에 따라 언어의 의미가 달라질 수도 있기 때문에 결합 방법은 매우 중요하다.

논리언어를 실행하는 단일화(unification)나 분해(resolution)는 함수언어를 구현하는 패턴매칭이나 감축에 비해 더 복잡하다. 따라서 함수 논리 언어의 실행을 위해 함수언어의 감축 방법을 확장하여 단일화 개념을 추가하는 것보다 함수를 감축할 수 있도록 단일화를 확장하는 것이 더 바람직하다[5]. 그 이유는 병렬처리의 효과를 높이기 위해서는 감축의 방법을 확장하는 것이 좋지만 단일화까지 처리하게되면 제어구조가 매우 복잡해져서 병렬처리로 얻는 장점이 감소되어 실행 효율면에서 별 소득이 없기 때문이다.

단일화에 함수 감축을 할 수 있도록 확장한 동등 단일화 알고리즘들은 지연 연산의 지원 여부와 함수내에서 논리 변수를 사용하는가의 여부에 따라 분류할 수 있다.

2.1 동등 단일화 알고리즘

동등 단일화 알고리즘은 두 항간의 가능한 모든 통합자(Unifier)를 찾기 때문에 많은 통합자나 무한한 통합자가 존재할 수 있어 매우 복잡하다. 따라서 함수 논리 언어의 표현력과 효율성은 선택한 동등 단일화 알고리즘에 따라 좌우된다. FUNLOG[13]에서 사용한 의미 단일화는 함수 항의 지연연산을 제공함으로써 무한자료구조를 처리할 수 있도록 하고 있으며, 함수 내에서의 논리 변수는 허용하지 않아 함수가 결정적으로 동작하도록 한다. Reddy의 언어를 위한 Narrowing은 무한자료구조는 지원하지 않으나 함수내의 논리 변수를 허용하고 있으며 Aflog언어를 위한 정규 단일화는 이거(eager) 연산을 하므로 무한 자료 구조를 지원하지 않고 함수의 결정적 실행을 위해 함수 내의 논리 변수도 허용하지 않는다.

본 논문에서는 데이터 플로우 모델에서의 실행에 적합하고 지연 연산을 지원하여 무한자료구조를 제공하며 함수의 결정적인 행동을 보장하기 위하여 함수 내에서 논리 변수를 허용하지 않는 동등 단일화 알고리즘을 사용하고자 한다.

알고리즘 1. 동등 단일화 알고리즘

Unify(a_n,b_n) {

input : a_n, b_n $\in$ {constant, variable, constructor, ...}*

output : binding list Q = $\bigcup_{i=1}^{n} \{X_i/m_i\}$

let V = {v | v is a variable symbol}

C = {c | c is a constructor}

F = {f(x_1, ..., x_n) | f is a function symbol}

process :

if (a_n $\in$ V) return(a_n/b_n)

if (b_n $\in$ V) return(b_n/a_n)

```
if (a_n = c(t_11, ..., t_1n) ∧ b_n = c(s_11, ..., s_1m))
    where c ∈C, {
  if (n ≠ m) return(failure)
  return(Unify(t_11,s_11) ∪ Unify(t_21,s_21)
        ∪ ... ∪ Unify(t_1n,s_1m)) }
if (a_n ∈ F ∧ b_n ∈ C)
if( Reduction(a_n) == failure) return(failure)
else return(Unify(Reduction(a_n), b_n)))
if (b_n ∈ F ∧ a_n ∈ C)
if( Reduction(b_n) == failure) return(failure)
else return(Unify(Reduction(b_n), a_n)))
}
Reduction(f_n) {
if (f_n ∈ C) return(f_n)
if (f_n ∈ F) {
let f_n = f(x_1, ..., x_n)
if (∃x_i, x_i is non-ground term)
  return(failure)
if (f is primitive function)
  return(evaluate(f_n))
if (there exist a rewrite rule f_n = g_n) {
  if (g_n is WHNF) return(g_n)
  else return(Reduction(g_n))
} } }
```

위의 알고리즘에서 함수는 절(clause)내의 서브 고울안에서 매개 변수로 표현되어 단일화 과정에서 감축된다. 이 때 함수가 감축되기 위해서는 모든 인자가 결정되어 있어야 하며 하나의 통합자 만이 생성되어 실행 결과를 반환한다. 만일 아직 그라운드되어 있지 않은 인자가 존재한다면 함수의 감축은 수행되지 않고 퇴각 검색이 일어난다. 이 알고리즘은 정규 단일화에 무한자료구조를 제공할 수 있도록 확장한 것이다.

2.2 구문 구조와 운영 의미

함수 논리 언어는 결합 형태에 따라 의미가 달라지고 운영 방법도 달라지기 때문에 새로운 실행 방법을 제시하고자하는 경우에 기존에 개발된 언어를 그대로 사용하기가 어렵다. 따라서 본 논문에서 사용하는 함수 논리 언어는 PROLOG와 HOPE를 기반으로 하고 있다. 그러나 논문의 목적이 함수 언어와 논리 언어를 결합하는 방법에 대한 것이 아니고 실행 방법에 관한 것이기 때문에 개략적인 구문 구조와 운영 의미만을 서술하고자 한다.

함수 논리 언어의 구문 구조를 BNF로 표현하면 다음과 같다.

```
program ::= function | clause
function ::= fct_head '===' fct_body '.'
fct_head ::= fct_symbol '(' fct_arg ')'
fct_body ::= term
  | 'IF' term 'THEN' term 'ELSE' term
  | '[' term [',' term]* ']'
fct_arg ::= term [',' term]*
clause ::= clause_head ':-' clause_body '.'
  | clause_head '.'
clause_head ::= pred_symbol '(' pred_arg ')'
clause_body ::= atom | atom ',' clause_body
pred_arg ::= term [',' term]*
atom ::= pred_symbol '(' term [',' term ]* ')'
  | term '=' term
term ::= variable
  | constructor_symbol '(' term [',' term]* ')'
  | fct_symbol '(' term [',' term]* ')'

fct_symbol ∈ F, constructor_symbol ∈ C,
variable ∈ V, pred_symbol ∈ P
```

여기서 C는 구조화된 자료 객체를 구성하기 위해 사용되는 함수 기호 형태의 "구성자"의 집합이고 F는 자료 객체의 계산을 수행하기 위해 사용되는 재기록(rewrite) 규칙의 이름을 구성하는 "함수 정의"의 집합이며 V는 변수의 집합이고 P는 술어(predicate) 기호의 집합이다.

위의 구문은 재기록 규칙의 합류(confluence)의 속성을 허용하기 위해 몇가지 제약 조건을 만족해야 한다. 우선, 모호성을 피하기 위해, 함수 fct_head_1과 fct_head_2가 선언되는 경우 이 두 함수는 단일화 되지 않아야 한다. 또한 왼쪽 선형도(left linearity)를 만족하기 위해 재기록 규칙의 왼쪽(입력패턴)에는 각 변수가 한번만 와야 한다.

본 논문에서 사용한 함수 논리 언어는 함수 언어와 논리 언어가 단독적으로 사용될 수 있으며 논리 언어 내에서 함수를 사용할 수 있다. 함수는 항상 서브고울의 인자 위치에 놓이게 되며 일차항으로 처리된다. 함수가 확장된 단일화 과정에서 평가됨으로써 언어의 논리적 배경은 동등성(Equality)를 가지는 혼 절(Horn clause)이다. 추론 메카니즘은 PROLOG에서 사용하는

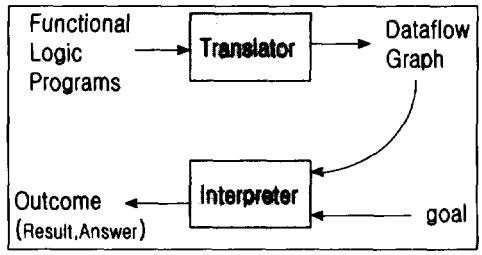
SLD분해와 바깥우선 감축(outermost reduction)을 사용하고 검색 방법은 실행의 단순성을 기하기 위해 왼쪽에서 오른쪽으로, 위에서 아래로 실행한다. 함수 내에서는 고차원 함수를 허용하고 감축은 WHNF까지만 수행한다.

3. 데이타 플로우 모델

입력이 준비된 연산자가 바로 실행되는 데이타 플로우 모델의 성질은 함수 논리 언어의 내재된 병렬성을 충분히 추출할 수 있는 기초를 마련한다. 따라서 이번 장에서는 함수 논리 언어를 효율적으로 수행하기 위해 각 절과 함수가 독립적인 데이타 플로우 그래프로 표현되는 모델을 디자인한다. 이들 각 그래프의 에지는 노드간의 데이타 패킷을 전송하는데 사용되고, 노드는 절이 실행될 때 수행될 기본 연산을 표현한다. 이 모델은 각 노드가 그래프내에서 동시에 처리되는 독립적인 계산을 허용한다.

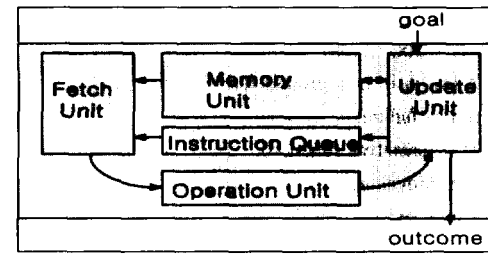
3.1 추상 기계의 구성

추상 기계는 규칙과 사실, 함수 등을 입력으로 받아 데이터 플로우 그래프로 변환하는 변환기(Translator) 부분과 데이터 플로우 그래프와 목적(Goal)을 입력으로 받아 결과를 생성하는 인터프리터 부분으로 나눌 수 있다. 변환기는 입력되는 사실, 규칙, 함수 등을 입력으로 받아 각각의 독립된 데이터 플로우 그래프를 구성한다. 인터프리터 부분은 Arvind[1]가 제안한 U-인터프리터와 유사한 구조를 갖는다. (그림1)에 추상 기계의 구조가 있다.



(그림 1) 추상 기계의 구조
(Fig. 1) Structure of Abstract Machine

인터프리터의 실행은 UPDATE 부분에서 고울과 지식 베이스를 가지고 시작한다. 입력된 고울로 지식 베이스(Memory Unit)를 수정하고, 지식 베이스 내에서 모든 입력이 준비된 노드를 명령 큐에 넣는다. FETCH 부분은 명령 큐에 있는 노드를 지식 베이스에서 호출한 후, 패킷을 구성하여 OPERATION 부분으로 넘긴다. OPERATION 부분은 패킷을 받아 연산자의 종류에 따라 연산을 수행하고 결과를 목적지 주소로 보내기 위해 UPDATE 부분으로 보낸다. 다시 UPDATE 부분은 생성된 결과를 지식 베이스에 반영하고 입력이 준비된 노드를 명령 큐에 넣는다. 명령의 실행이 (FETCH, OPERATION, UPDATE)의 순으로 순서적으로 동작하기 때문에 이는 환형 파이프라인과 같이 동작한다. (그림 2)에 인터프리터의 구조가 있다.



(그림 2) 인터프리터의 구조
(Fig. 2) Structure of Interpreter

3.2 패킷의 구조

패킷은 두 개의 기본적인 연산기 사이에 전달되는 메시지로서 FETCH 부분은 지식 베이스에서 로드한 내용을 연산자와 목적지 주소등으로 구성하고, OPERATION 부분은 실행 결과인 바인딩 리스트와 리터럴 데이타 등을 추가한다. UPDATE 부분은 받은 패킷을 이용하여 지식 베이스를 수정하고 입력이 준비된 노드의 주소를 명령 큐에 넣는다. 패킷의 구조는 표 1과 같다.

〈표 1〉 패킷의 구조
〈Table 1〉 Structure of Packet

색	반환 주소	목적지 주소
퇴각검색 주소	바인딩 리스트	리터럴 데이타
연산자		

색은 그래프 내에서 각 노드의 병렬 수행을 위해 현재 수행하는 플로우를 구분하기 위한 것이고, 반환 주소는 결과를 되돌릴 그래프의 번호이다. 목적지 주소는 연산의 수행 후, 보낼 그래프 번호와 노드 번호,

포트 번호로 구성되고, 퇴각 검색 주소는 실행을 실패할 경우 재 수행할 주소이다. 연산자는 수행할 연산의 종류를 지정하고 바인딩 리스트는 논리 언어의 결과를 반환하기위해 각 논리 변수의 바인딩 상태를 유지하며 리터럴 데이터는 함수의 감축 결과를 유지한다.

3.3 지식 베이스의 구조

함수 논리 언어를 입력으로 받아 생성되는 데이터 플로우 그래프는 지식 베이스의 형태로 기억 장소에 저장된다. 실행에 필요한 지식 베이스의 구성 요소는 표 2에 있다.

〈표 2〉 지식 베이스의 구조
〈Table 2〉 Element of Knowledge

색	플래그	필요한 입력의 수
도착한 입력의 수	연산자	목적지 주소
퇴각검색 주소	상태	입력 리터럴
입력 바인딩 리스트	바인딩 리스트	리터럴 데이타

플래그는 필요한 입력의 수와 도착한 입력의 수가 같게 되면 1로 되어 실행할 수 있는 상태임을 나타낸다. 상태 필드는 처음에 초기 상태에 있다가 첫 번째 서브 고울을 실행 중이라면 대기 상태에 있고 첫 번째 서브고울을 수행한 후, 다른 후보해를 수행 중이면 실행 상태로 바뀐다. 만일 모든 후보해를 계산하여 더 이상 실행할 후보해가 없는 경우에 종료 상태가 된다. 입력 리터럴과 입력 바인딩 리스트는 현재 도착한 입력을 유지 하고 바인딩 리스트와 리터럴 데이터는 현재 노드의 상태와 결과를 반환하기 위해 사용된다.

3.4 기본 명령어

데이터 플로우 모델의 기본적인 명령어들은 함수 언어 부분을 위한 명령과 논리 언어 부분을 위한 명령으로 나눌 수 있다. 함수 언어를 위한 명령들은 인자를 분배하여 병렬 처리할 수 있도록 하는 DISTRIBUTE명령과 IF-THEN-ELSE의 조건문을 위한 SWITCH, MERGE문, 또 다른 함수의 호출을 위해 인수를 모으는 ASSEMBLE명령과 APPLY명령 등이 있고, 논리 언어 부분을 위한 명령으로는 서브고울의 실행을 관리하는 EXECUTE와 실행 결과를 반영하기위한 UPDATE 명령등이 있다. 이 밖에 함수나 규칙, 사실 등을 수행하기 위해 단일화를 수행하는 UNIFY명령과 계산된 결과를 되돌리는 RETURN문, 리스트의 처리를 위한 CONS명령 등이 있다.

UNIFY명령은 항 단위의 단일화를 수행하고 단일화가 성공하면 바인딩 리스트를 출력하고 그렇지 않으면 실패 메시지를 보내 퇴각검색이나 다른 후보해를 수행하도록 한다. 항의 형태는 함수나 구성자의 형태가 가능하나 명령의 단순화를 위해 그래프 구성 과정에서 세분하여 처리하도록 하고, UNIFY명령은 변수와 상수에 대한 처리만을 하도록 한다. UNIFY명령에 대한 개략적인 행동은 다음과 같다.

```
UNIFY(a, b) {
  if (a == variable && b == constant)
     send_next(a/b);
  if (b  == variable && a == constant)
     send_next(b/a);
  if (a = constant && b == constant)
     if (a = b) send_next(success)
     else send_previous(failure)
}
/* send_next(A) : A를 목적지 주소로 보냄 */
/* send_previous(A) : A를 퇴각검색 리스트의 top에
있는 주소로 보냄 */
```

DIST_HEAD는 구성자와 인자를 분리하여 단일화 작업을 세분화하는데 사용된다. FUNCTION은 절의 인자 부분에 발생하는 함수를 단일화하기 위해 미리 계산해야 하는 경우에 사용하는 명령어이다. EXECUTE는 서브고울의 실행을 관리하는 명령으로 처음에는 초기 상태에 있다가 입력이 도착하면 상태를 대기로 바꾸고 첫 번째 후보해를 실행하기 위해 메시지를 보낸다. 첫 번째 후보해의 결과가 도착하면 상태를 실행으로 바꾼다. 이후에 퇴각 검색에 의해 다른 후보해의 연산을 요구하면 다시 상태를 대기로 바꾸고 다음 후보해를 실행한다. 더 이상 실행할 후보해가 없으면 상태를 완료로 바꾼다. 이 후에 재실행 명령이 도착하고, 현재 상태가 완료이면 퇴각검색 주소와 함께 실패 메시지를 되돌리고 현재 상태를 초기로 바꾼다.

```
EXECUTE(subgoal) {
   if (state == initial) {
      state = wait;
```

```
        result = call (subgoal);
        send_next(result)
        if (exist more subgoal)
            state = execute;
        else state = finish;
    }
    else if(state == execute) {
        state = wait;
        result = call(another subgoal);
        send_next(result)
        if (exist more subgoal)
            state = execute;
        else state = finish;
    }
    else
        send_previous(fail);
}
```

APPLY명령은 함수의 몸체를 실행할 때 또다른 함수나 자기 자신을 다시 부르는 경우에 사용한다. 만일 APPLY한 결과가 프리미티브 연산이라면 이를 수행한 결과를 출력하고, 그렇지 않으면 함수와 인자를 붙인 형태를 그대로 출력한다. 이것은 지연연산을 수행하기 위한 것으로 여기서 함수를 끝까지 계산한다면 이거(eager)연산을 하게 된다.

```
APPLY(fun, arg_list) {
    if (fun == primitive function)
            send_next(compute(fun arg_list));
    else  send_next(fun(arg_list));
}
```

UPDATE명령은 두 입력 포트에 도착하는 바인딩 리스트를 서로 적용하고, 두 번째 입력의 리터럴 데이터를 반영하며 두 입력의 퇴각 검색 리스트를 하나로 결합하는 작업을 하게 된다. DISTRIBUTE명령은 함수를 위한 그래프를 구성할 때 사용하는 명령으로 함수의 병렬 감축을 위해 인자들을 나누어 분산하는 일을 한다. ASSEMBLE명령은 함수의 몸체내에서 다른 함수를 호출하는 경우, 이에 필요한 인자를 모으는 작업을 한다. 인자들은 DISTRIBUTE명령에 의해 분산된 인자들이나 혹은 이들을 기반으로 계산된 결과가 된다. SWITCH명령은 IF-THEN-ELSE의 조건문을 수행하기위해 두 개의 입력과 두 개의 출력을 갖는다. 두 번째 입력은 부울값을 가지며 이 값에 따라 참이면 첫 번째 출력으로, 거짓이면 두 번째 출력으로 첫 번째 입력을 내보낸다. MERGE명령은 IF_THEN_ELSE 문을 위해 분기된 결과를 하나로 모으기 위해 사용한다. 3개의 입력과 1개의 출력을 갖는데, 3번째 입력은 부울값으로 참이면, 첫 번째 입력을 출력포트로 보내고 거짓이면 두 번째 입력을 출력포트로 보낸다. COPY 명령은 입력으로 들어오는 바인딩 리스트와 리터럴 데이터를 출력 포트로 보낸다. RETURN명령은 입력을 받아 원래 호출한 노드로 결과를 되돌릴 때 사용한다. 만일 반환주소가 없다면 전체 해가 구해진 상태로 수행이 종결된다. CONS명령은 리스트의 처리를 위해 사용되는 명령으로 원자(atom)와 리스트를 연결하여 하나의 리스트로 만든다. FIRST명령은 리스트를 입력으로 받아 리스트의 첫 번째 원자를 출력하는 것으로 리스트 처리에 사용된다. REST명령은 리스트를 입력으로 받아 리스트의 첫 번째 원자를 제외한 나머지 리스트를 출력하기 위해 사용된다. CONST명령은 상수 값을 표현하기 위해 사용한다.

4. 그래프 구성 방법

앞 절에서 설명한 각 명령어를 사용하여 함수 논리 언어를 데이터 플로우 그래프로 표현할 수 있다. 사실이나 규칙, 함수 등을 각각의 독립적인 그래프로 생성한다. 노드는 명령어를 나타내며, 에지는 명령간의 메시지를 전달하는 것이다. 그래프 구성 알고리즘은 크게 논리 언어 부분을 위한 절 변환 알고리즘과 함수 언어 부분을 위한 함수 변환 알고리즘으로 분류할 수 있다.

4.1 절(predicate) 변환 알고리즘

규칙이나 사실을 위한 그래프는 입력으로 들어오는 패킷과 노드안에 존재하는 서브고울간의 UNIFY명령으로 시작한다. 우선, 절의 이름을 UNIFY하고, 각 인자들을 다시 각각 UNIFY명령으로 단일화 한다. 만일 인자 중에 함수명이 있으면 감축을 먼저 실행하고 나서 그 결과를 단일화한다. 단일화가 성공하면, 바인딩 리스트를 가지고 절의 몸체 부분을 수행하기 위한 코드를 생성한다. COPY명령으로 바인딩 리스트를 복사하고 각 서브고울들은 UPDATE명령으로 바인딩 상태를 입력으로 받아 실행한다. 각 서브고울의 결과를

UPDA-TE명령을 통해 받고 바인딩이 성공적으로 수행되면 RETURN명령을 이용해 결과를 되돌린다.

```
let predicate = form of head :- subgoals.
let head = form of p(x1, x2, ... xn),
    input goal = q(y1, y2, ..., ym).
COMPILE [predicate] =
    DIST_HEAD(goal, head) [0, 1, 2, ... , n]
      0:   UNIFY(q, p) [c.0]
      1:   COMPILE_UNIFY[y1, x1]
           ...
      n:   COMPILE_UNIFY[yn, xn]
      c:   MERGE_HEAD(_, _, ..., _) [d.1]
      d:   COMPILE_SUB[subgoals]
COMIPLE_UNIFY[xi, yi] = if (yi = function) {
            FUNCTION(xi) [q.1]
            q: UNIFY(_, yi) [c.i]  }
            UNIFY(xi, yi) [c.i]
let subgoals = form of subgoal1, ..., sungoaln
COMPILE_SUB[subgoals] = COPY (1,2, ..., n)
      1: UPDATE (subgoal 1)
         COMPILE_FUN_ARG[subgoal 1]
         EXECUTE
      r: UPDATE r2
           . . .
      n: UPDATE (subgoal n)
         COMPILE_FUN_ARG[subgoal n]
         EXECUTE
    m-1: UPDATE m
     m: RETURN
let subgoal = form of q(X1, X2, ..., Xn)
COMPILE_FUN_ARG [subgoal] = for(i=1; i≤n;i++)
      if (Xi = function term)
         FUNCTION(Xi)
```

4.2 함수 변환 알고리즘

함수를 위한 그래프 구성 알고리즘도 UNIFY에서 시작한다. 함수의 모든 인자가 그라운드 되어 있지 않다면 실패를 반환하고 그렇지 않으면 함수의 몸체를 감축한다. 함수의 몸체를 구성하기 위해 DISTRIBUTE 명령으로 함수의 각 인수를 병렬 처리하기 위해 분리하고, 조건문을 위해 SWITCH명령과 MERGE 등을 이용한다. 또한 다른 함수를 호출하여 수행하는 경우 ASSEMBLE로 인자를 모으고 APPLY로 실행한다.

```
let function = form of fun_head === fun_body.
let fun_head = form of f(x1, x2, ... xn),
               input = g(y1, y2, ..., ym).
COMPILE_FUN [function] =
      DIST_HEAD(input, fun_head) [0, 1, 2, ... , n]
   0: UNIFY(g, f) [c.0]
   1: COMPILE_UNIFY_FUN[y1, x1]
           ...
   n: COMPILE_UNIFY_FUN[yn, xn]
   c: MERGE_HEAD(_, _, ..., _) [d.1]
     DISTRIBUTE(arguments)
     COPY
     COMPILE_BODY[fun_body]
COMIPLE_UNIFY_FUN[xi, yi] =
      if( yi is not ground value)
            send(failure) [c.i]
      else {
         if (yi = function) {
              FUNCTION(xi) [q.1]
          q: UNIFY(_, yi) [c.i] }
         UNIFY(xi, yi) [c.i]   }
COMPILE_BODY[fun_body] =
     if (fun_body = constant)
         CONST(fun_body)
     if (fun_body is form of "if c then p1 else p2") {
         COMPILE_BODY(c)
         SWITCH r1 r2
     r1: COMPILE_BODY[p1]
     r2: COMPILE_BODY[p2]
         MERGE                    }
     if (fun_body is form of "f(x1, ..., xn)" {
        ASSEMBLE(x1, ..., xn)
        APPLY(f, (x1, ..., xn))    }
     if (fun_body is form of "[H|T]" {
        ASSEMBLE(H, T)
        CONS       }
```

4.3 제한된 AND-병렬성의 도입

하나의 고울(goal)을 해결하기 위해 여러개의 서브고울을 수행해야 하는 경우 이들 각각의 서브고울은 함수 논리 언어가 가지는 비결정성의 성질에 따라 수행 순서에 무관하므로 동시에 수행될 수 있다. 그러나 각 서브고울이 같은 변수를 공유하고 있다면, 공유변수는 유일하게 바운드되어야 하므로, 이러한 공유 변수가 없어야만 병렬수행이 가능해 진다. 이와 같은 병

렬성을 독립 AND-병렬성이라 한다.

공유 변수가 존재하는 경우에 독립 AND-병렬성을 구현하기 위해서는 변수를 공유하는 서브고울들을 동기화 시켜야 한다. 동기화는 수행 시점에 따라 정적 종속성 분석, 동적 종속성 분석, 제한된 AND-병렬성 등으로 나눌 수 있다. 정적 종속성 분석은 컴파일 시간에 종속성이 분석되므로 실행의 추가부담이 전혀 없으나 프로그램에 내재된 병렬성을 제대로 추출할 수 없다는 단점이 있다. 동적 종속성 분석 방법은 실행시간에 종속성 분석이 이루어 지므로 프로그램이 가지는 병렬성을 최대한 추출할 수 있으나 추가 부담이 너무 크다. 제한된 AND-병렬성(RAP)은 위의 두가지 방법의 장점을 조합하는 것으로 2단계로 나눌 수 있다. 우선, 컴파일 시간에 서브 고울간의 간단한 종속성 환경이 만들어 지고, 실행 시간에 검사해야할 조건 연산식이 만들어 진다. 실행 시간에 미리 만들어진 조건 연산식을 수행하면서 각 서브 고울의 병렬 실행 여부를 결정한다. 이것은 단순하며 많은 시간을 요하지도 않으나 구조화된 리스트의 처리와 같은 경우에 일부 AND 병렬성을 잃을 수도 있다.

제한된 AND-병렬성을 우리의 알고리즘에 도입하기 위해서는 서브고울간에 병렬 수행이 가능한가를 다음과 같은 순서로 종속성 검사를 해야 하고, 종속성이 존재하는 경우 이들을 동기화시켜야 한다.

① 서브고울이 지역변수를 공유하는 경우

서브고울간에 공유하고 있는 변수가 입력 변수가 아니면 무조건 순차실행을 한다.

② 서브고울이 입력변수를 공유하는 경우

서브고울이 입력변수를 공유하고 있다면 위의 경우와 마찬가지로 종속적으로 수행되어야 한다. 그러나 만일 입력변수에 값이 바인드되어 들어온다면 두 서브고울은 병렬로 수행할 수 있다. 따라서 입력 변수 P에 대한 그라운드 검사가 필요하다.

③ 서브고울이 공통변수는 갖지 않으나 입력변수를 나누어 갖는경우

두 서브고울이 공통변수를 갖지 않으면 병렬로 두 개의 서브고울을 수행할 수 있다. 그러나 입력 변수를 나누어 갖고 있다면 질의의 형태에 따라 순차 수행을 해야하는 경우가 발생한다. 예를들면,

possible_pair(X,Y) :- boy(X), girl(Y).

이 문장에서 두 개의 서브고울 boy와 girl은 병렬로 수행할 수 있다. 그러나 질의가 :- possible_pair(A,A)와 같은 형태로 들어온다면 이는 입력변수를 공유하는 경우와 마찬가지로 순차적으로 수행해야 한다. 따라서 입력변수 X와 Y가 서로 독립적인가 하는 독립 검사가 필요하다.

④ 서브고울이 입력변수를 공유하고, 입력변수를 나누어 갖는 경우

두 서브고울이 ②항과 ③항의 경우에 둘다 해당되는 경우에는 공통변수간의 바인딩검사와 두 입력변수간의 독립성을 검사해야 한다. 검사 결과에 따라 병렬 수행의 여부가 가려진다. 예를 들면,

parent(C,M,F) :- mother(C,M), father(C,F).

이 예에서 두 개의 서브고울 mother와 father가 입력변수 C를 공유하면서, 다시 입력변수 M과 F를 나누어 갖는다. 이 경우에는 변수 C가 질의에서 어떤 값으로 바운드되었는가를 검사한 후, 바운드 되었다면, 다시 변수 M과 F가 서로 독립적인가를 검사하여 독립적이라면 병렬 수행이 가능하고, 그렇지 않은 모든 경우에는 순차 수행을 하도록 해야한다.

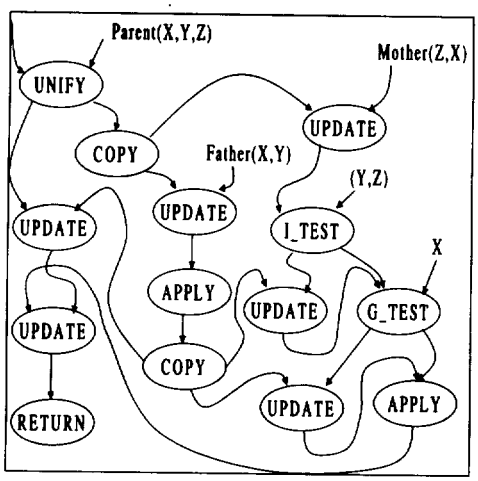
⑤ 서브고울이 공통변수를 갖지 않고, 입력변수를 나누어 갖지도 않는 경우

두 서브고울이 공통변수가 없고 입력변수도 나누어 갖지 않는 경우에는 두 서브고울이 아무런 조건 없이 병렬로 수행할 수 있으므로 알고리즘의 변화는 없다.

• I_TEST와 G_TEST 명령어의 추가

종속성 검사를 위해 I_TEST와 G_TEST를 명령어 집합에 포함하자. I_TEST는 서로 다른 두 입력 변수간의 바운드 상태를 검사하기 위한 것으로 두 개의 입력과 2개의 출력을 갖는다. 따라서 다른 두 입력 변수가 바운드되어 있으면 두 번째 출력으로 결과를 보내고, 그렇지 않으면 첫 번째 출력으로 결과를 보낸다. G_TEST는 입력 변수를 서브고울들이 나누어 갖는 경우 변수의 그라운드 상태를 검사하기 위한 것으로 두 개의 입력과 두 개의 출력을 갖는다. 입력 변수가 바운드되어 있는 경우에는 두 번째 출력으로 결과를 보내고 바운드되어있지 않은 경우에는 첫 번째 출력으로

결과를 보낸다. 그림 3은 위의 종속성 분석에서 ④번 항에 대한 내용을 구현한 것이다.



(그림 3) I_TEST와 G_TEST의 사용예
(Fig. 3) Example of I_TEST and G_TEST

4.4 병렬 감축

함수는 서브고울의 인수 부분에 위치하여 단일화를 하는 과정에서 감축된다. 이 과정에서 함수의 인수가 모두 그라운드 되어 있으면 감축이 가능해지고, 감축된 결과를 단일화에 사용하게 된다. 각 함수는 처치-로서(Chursh-Rosser)의 정리에 의해 수행 순서에 무관하게 같은 결과를 얻을 수 있으므로 병렬 실행에 적합하다. 여기서 함수의 감축을 WHNF까지만 수행하면 무한자료구조를 지원할 수 있게된다. 따라서, 우리의 모델에서 함수는 WHNF까지만 감축되고, 자연스럽게 병렬 수행이 가능하다.

5. 실험 결과 및 분석

5.1. 관련 연구와의 비교

Degroot가 제안하고 Hermenegildo[7]에 의해 개선된 CGE (Conditional Graph Expression)는 논리 언어의 병렬성을 효과적으로 표현할 수 있다. 따라서 많은 연구에서 함수 논리 언어의 병렬성을 표현하기 위해 CGE 표현 방법에 함수를 추가한 CGE+[8][13]표현 방법을 사용하였다. 본논문에서 제시한 DFG는 함수 논리 언어안에 내재된 병렬성을 자연스럽게 추출할 수 있고 CGE+ 표현 방법에 비해 지연 시간을 줄일 수 있으므로 더 효율적이다. 즉, CGE+ 표현법에서는 종속성의 검사를 서브 고울을 수행하기 전에 실시하고 이 결과를 가지고 순차 수행을 할 것인가 아니면 병렬 수행을 할 것인가를 결정한다. 만일 순차 수행을 해야하는 경우라면 종속성을 검사한 시간 만큼 지연 시간이 발생하게 되고, 병렬 수행을 하는 경우에도 종속성을 검사하는 시간은 추가 지연 시간이 된다. 우리의 DFG는 종속성의 검사와 첫 번째 서브고울을 병렬로 처리하기 때문에 종속성 검사로 인한 추가 시간은 전혀 없게 된다.

다음의 예를 보자.

예 1) p(X) :- (ground(X) | q(X) & r(X))

예 2) f(X) ==> (X==0) | 0,
+(PAR fib(X) fib(2*X)).

예 1)과 2)는 CGE+로 표현된 예이다. ground(X)는 변수 X가 그라운드되어 있는가를 서브고울들이 수행되기 전에 검사하는 부분으로 X에 대한 검사를 해 보아야 q와 r을 순차 실행 할지 아니면 병렬 수행 할지 결정할 수 있다. 또한 예 1)의 ground(X)와 예 2)의 PAR는 종속성 검사와 병렬 실행 여부를 사용자가 명시해야 하므로 사용자에게 부담을 주게되고 오류 발생의 가능성도 높아지게 된다. 그러나 본 논문에서 제시한 DFG에서는 ground(X)와 q(X)를 병렬로 실행하고 ground(X)의 결과에 따라 바로 r(X)를 수행할 것인지 아니면 q(X)의 수행이 끝난 뒤에 수행할 것인지를 결정하기 때문에 지연시간이 없다. 또한 이와 같은 종속성의 검사나 병렬 처리의 여부를 자동으로 추출하여 실행하므로 사용자에게 부담을 주지 않는다.

5.2 성능 분석

함수 논리 언어를 데이터 플로우 그래프로 표현하는 방법의 효율성을 보이기 위해 기존에 논리 언어의 병렬성 검사를 위해 사용된 몇개의 테스트 프로그램을 도입한다. Check-N 프로그램은 hermenegildo[7]가 제시한 10개의 서브 고울을 병렬 처리 가능한 요소로 두고 각각의 서브 고울을 N번 반복처리하는 프로그램을 변형하여 각 각의 서브 고울이 2*N번 반복하여 되부름을 처리하도록 한 것이다. 이 프로그램을 도입하여 변형한 이유는 많은 병렬성을 갖는 프로그램에서 종속성 분석을 위한 지연 시간을 분석하기 위한 것이다. Fibonacci-20 프로그램은 20개의 피보나치 수를 찾아 출력하는 프로그램이고, Lucky-Number 프로그램은 1부터 시작하는 홀수 중에서 자신의 숫자에 해당하는

차례에 나오는 숫자를 제거하고 남는 숫자만을 찾는 프로그램으로 고차원(Higher Order) 함수의 표현과 지연 연산을 통한 무한 자료 구조를 처리하기 위한 것이다.

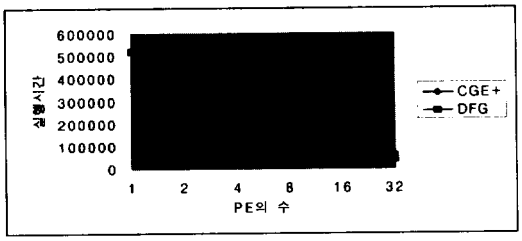
병렬 실행 모델의 성능을 분석하기 위해 본 논문에서는 각 데이터 플로우 그래프를 각각의 PE로 분산시키면서 그래프 내의 연산은 각 PE내에서 수행하도록 하는 서브 고울과 함수 단위의 병렬 처리를 채택하였다. 그 이유는 명령어 단위의 병렬성을 이용하여 처리하는 경우 발생하는 통신의 추가 부담을 줄이기 위한 것이다. 본 실험의 목적이 데이터 플로우 모델이 함수 논리 언어가 가지고 있는 병렬성을 얼마나 자연스럽게 표현하고 효율적으로 처리하는가에 대한 것이므로, 데이터 플로우 그래프의 코드를 각 PE로 이동시키는 시간이나 방법은 고려하지 않았고, 가베지 수집(Garbage Collection)과 같은 메모리 관리 문제도 고려하지 않았다. 또한 네트워크에서 발생하는 통신상의 추가 부담도 고려하지 않았다.

5.3. CGE+와 DFG의 지연시간 비교

CGE+ 표현 방식은 우선 종속성을 검사한 후, 병렬 실행 또는 순차 실행을 한다. 본 논문의 DFG 표현 방식은 종속성의 검사와 첫 번째 서브 고울을 동시에 처리하여 종속성 검사를 위한 추가 부담을 갖지 않는다. 따라서 모든 종속성이 발생하는 곳에서 지연 시간의 차이가 나게 된다. 다음은 논리 언어의 병렬성을 분석 비교하기 위해 Hermenegildo에의해 사용된 벤치마크 프로그램인 Check-10을 통하여 CGE+와 DFG의 실행 속도 및 성능을 본 논문의 병렬 추상 기계하에서 PE의 갯수를 늘려 가며 실험한 결과이다. 모든 조건은 동일한 상태에서 실험하였고, 통신 상의 추가 부담이나 가베지 수집과 코드 이동과 같은 메모리 관리 측면을 고려하지 않았으므로 실행 시간의 비교는 각 명령어가 동일한 처리 시간을 갖는다고 가정하고 전체 명령어의 수행 빈도로 측정하였다.

● 실행 속도의 비교

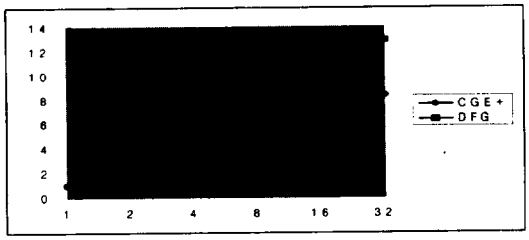
Check-10 프로그램은 풍부한 병렬성을 갖는 프로그램으로 PE의 수를 늘려 감에 따라 CGE+보다 DFG의 실행 시간이 PE의 수에 비례하여 감소하는 것을 예측해 볼 수 있다. 그 이유는 CGE+ 표현에서의 종속성 검사로 인한 지연 시간을 DFG에서는 줄일 수 있기 때문이다.



(그림 3) CGE+와 DFG의 실행 속도
(Fig. 3) Comparison of execution speed for CGE+ and DFG

● 성능 향상도 비교

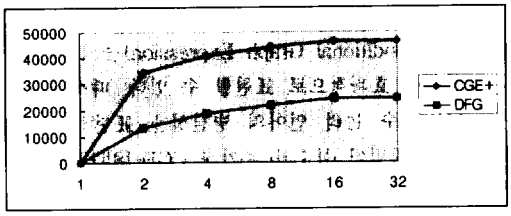
그림 4에 CGE+와 DFG간에 PE의 수에 따른 성능 향상도의 비교가 있다. PE의 수를 8개로 늘렸을 때 CGE+표현의 경우는 약 5배의 성능 향상의 효과가 있지만 DFG의 경우에는 약 6배 정도의 성능이 향상되는 것을 볼 수 있다. 이 차이는 PE의 수가 늘어남에 따라 점점 더 차이가 나는 것을 볼 수 있다.



(그림 4) 성능 향상도 비교
(Fig. 4) Comparison of Speed-Up

● 평균 지연 시간의 비교

PE의 수를 늘려 감에 따라 각 PE에서 실행을 하지 못하고 기다리는 시간이 늘어나게 된다. 그림 5는 Check-10 프로그램의 실행과정에서 PE의 수를 늘려감에 따라 각 PE에서 발생하는 평균 지연 시간을 비교한 것이다. 각 PE에서의 평균 지연 시간이 DFG에 비해 CGE+에서 2배정도 발생하는 것을 볼 수 있다.



(그림 5) 평균 지연 시간의 비교
(Fig. 5) Comparison of Average Delay Time

6. 결 론

지난 10여년 동안 함수 언어와 논리 언어를 하나의 시스템에서 동시에 제공하기 위해 많은 연구가 있었다. 이들 연구의 주 목적은 전체 시스템의 성능을 저하시키지 않으면서 각각의 프로그래밍 스타일을 적절하게 제공하여 두가지 프로그래밍 언어의 장점을 얻고자 하는 것이다. 그러나 이와같은 목적을 이룬 연구 결과는 아직 거의 없다. 풍부한 함수나 다양한 선언적 언어의 특징을 제공하는 언어는 결합한 모델의 성능이 만족스럽지 못하고, 실행 성능이 효율적인 모델들은 함수 언어나 논리 언어의 중요한 특징을 모두 제공하지 않는다.

이 논문의 목적은 함수 논리 언어를 효율적으로 실행하기 위한 방법을 제시하기 위한 것으로 지연 연산을 제공하는 동등 단일화를 제시하여 함수 언어와 논리 언어가 가지는 장점을 모두 수용하고 이를 데이터 플로우 그래프로 표현하여 내재된 병렬성을 자연스럽게 추출하고 종속성 분석 등의 병렬 실행 기법을 데이터 플로우 그래프로 표현하는 방법을 제시하였다. 함수 논리 언어의 병렬성을 표현하는 기존의 CGE+ 방법과의 비교를 통하여 데이터 플로우 그래프 표현방식이 더 효율적임을 보였고, 병렬 실행 모델로 이를 확장하여 성능 분석을 하였다.

- 앞으로의 연구 방향

실행 모델의 단순성을 기하기 위하여 서브 고울 간의 종속성 검사에서 항상 서브 고울이 나오는 순서를 기준으로 규칙 검색 방법을 왼쪽에서 오른쪽으로 하였으나, 이는 불필요한 퇴각 검색을 유도할 수 있으므로 이에 대한 연구가 더 이루어져야 한다. 또한, 실행 모델에서 코드의 이동방법이나 가베지 수집에 대한 부분 등 메모리 관리 문제를 고려하여 실제 멀티프로세서 시스템에서의 구현이 앞으로의 남은 연구 과제이다.

참 고 문 헌

[1] Arvind, D.E. Culler, *Dataflow Architectures*, Annual Reviews in Computer Science, pp.1:225-253, 1986.

[2] I. Aybay, M. Baray, "An OR-Parallel and Restricted AND-Parallel, Nonbacktracking Prolog Execution Model", *Proceeding of IEEE 5th International parallel processing Symposium*, pp. 642-645, 1991.

[3] E.Borger, F.J.Lopez-Fraguas, M.R. Rodriguez-Artalejo, "A model for mathematical analysis of functional logic programs and their implementations", *IFIP 13th World Computer Congress, Volumn I: Technology/Foundations*, 1994.

[4] J.S. Conery, The AND/OR *Process Model for Parallel Interpretation of Logic Programs*, phD Dissertation, Univ. of California, Irvine, 1983.

[5] J. Darlington, A.J. Field, and H. Pull, "The Unification of Functional and Logic Language", *LOGIC PROGRAMMING Functions, Relations, and Equations*, pp.37-70, 1986.

[6] A.J. Field, P.G. Harrison, Functional Programming, Addison Wesley, 1988.

[7] M.V. Hermenegildo, *An Abstract Machine based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, phD Dissertation, Dept. og CS, Univ. of Texas at Austin, 1986.

[8] H. Kuchen, R. Loogen, J.J.Moreno-Navarro, M. Rodriguez-Artalejo, *Graph-Based Implementation of a Functional Logic Language*, Technical Report AIB 89-20, RWTH Aachen, Lehrstuhl f, 1989.

[9] J.W. Lloyd, "Combining Functional and Logic Programming Languages", *Proceeding of the 1994 International Logic Programming Symposium*, 1994.

[10] J.J. Moreno-Navarro, Expressivity of Functional -logic Languages and their implementation, *Invited tutorial at Joint International Conference on Declarative Programming*, 1994.

[11] G. Nadathur, B. Jayaraman, "Towards WAM Model for λProlog", Logic Programming: *Proceeding of the North American Conference*, pp. 1180-1198, 1989.

[12] D.W. Shin, A Types Functional Logic Language: Semantics and Implementation, phD Dissertation, Dept. of CS, KAIST, 1990.

[13] P.A. Subrahmanyam, J.H. You, "FUNLOG:A

Computational Model Integrating Logic Programming and Functional Programming", *LOGIC PROGRAMMING Functions, Relations, and Equations*, pp.157-198, 1986.

김 용 준

1989년 동국대학교 수학과(이학사)
1991년 동국대학교 대학원 컴퓨터공학과(공학석사)
1996년 동국대학교 대학원 컴퓨터공학과(공학박사)
1992년~현재 김천대학 조교수
관심분야 : 함수언어를 위한 컴퓨터 시스템과 활용 언어 병렬처리, 분산처리

전 서 현

1978년 경북대학교 공학사
1980년 한국과학 기술원(공학석사)
1991년 한국과학 기술원(공학박사)
1980년~1983년 계명대학교 전임강사
1983년~현재 동국대학교 컴퓨터공학과 교수
관심분야 : 함수언어를 위한 컴퓨터 시스템과 활용 언어시스템 통합