창 립
40주년학술대회
논문87-1-20-2

계통 사고 복구 전문가 시스템에서의 수치 데이타 처리
- IBM PC 용 Turbo prolog 에서 -

°최 준 영* 박 인 규* 박 종 근*
*서 울 대 학 교

Numerical data processing on expert system for power system fault restoration
- in IBM PC Turbo prolog -

Joonyoung Choi* Ingyu Park* Jongkeun Park*
*Seoul National University

ABSTRACT - This paper deals with expert system for power system fault restoration and accompanying numerical data processing. Nowadays, expert system which is a branch of artificial intelligence expands its application area to many fields. And it requires computer language for A.I. to be versatile. Expert system for power system handles numerous numerical data and language for A.I. has its deficiency in numerical data processing. However some recent version of the A.I. language find ways of overcoming this dilemma by giving the way of linking conventional algorithmic languages to them. This study presents numerical data processing routines described in Turbo prolog which is run in IBM PC and linking numerical data processing routines written in Turbo C to Turbo prolog.

서 론

인간의 지식중에는 알고리즘으로 표현하기가 곤란한 것들이 많이 있다. 이러한 지식을 Fortran이나 Pascal과 같은 언어로 기술하는 것은 매우 비효율적인 것이 된다. 인공지능용 언어는 이렇게 알고리즘으로 표현하기가 곤란한 지식들을 컴퓨터에 쉽게 적용할 수 있도록 만들어진 언어이다.

최근 전문가 시스템기법의 응용이 여러 분야로 확장되어감에 따라 인공 지능용 언어에는 그 본래적인 기능외에 알고리즘적 프로그램도 수행할 수 있는 기능이 요구되고 있다. 계통사고시 복구 전문가 시스템에서도 마찬 가지로 전문가의 지식을 담을 수 있는 구조 외에 수치 데이타 처리가 중요한 부분을 차지한다.

현재의 인공지능용 언어는 수치 데이타 처리 능력에 미비한 점이 많은데, 최근 필요성이 대두됨에 따라 그러한 기능의 추가되거나 수치 처리용 언어인 Fortran, C등을 연결하는 기능이 첨가되고 있다. 본 연구에서는, IBM PC용 Turbo prolog를 대상으로, prolog로써도 Fortran과 유사한 효율을 갖는 수치 데이타 처리 routine이 개발될 수 있음을 보였다. 그리고 prolog와 다른 언어들과의 연결에 관하여 논하였다.

본 론

1. prolog내에서의 수치 데이타 처리

prolog는 원래 인공지능 기법을 구현하기 위하여 만들어진 언어이다. 그렇기 때문에 prolog가 비록 기본적인 산술 연산 기능과 내장함수 들을 가지고 있기는 하지만, 원칙적으로 prolog는 수치 데이타 처리용이 아니다. 그러나 prolog로도 Fortran과 동일한 기능을 수행하는 프로그램이 개발될 수 있다.

$$I = \int_0^{\pi} \sin(x)dx$$

이 정적분을 trapezoidal rule에 의해 계산하는 문제를 예로 들었다.

그림 1 은 Fortran을 사용하여 프로그램을 구현한 예이며 그림 2 는 이 Fortran 프로그램과 동일한 기능을 하는 prolog 프로그램의 예를 보인 것이다. DO loop의 구조가 prolog에는 없지만 프로그램에서 보는 바와 같이 recursion을 이용하여 DO loop와 동일한 기능을 하는 routine을 개발할 수 있다. 이 예에서 알 수 있는 바와 같이 인공지능 문제와는 반대로 수치계산의 문제는 prolog가 Fortran보다 일반적으로 더 복잡하다.

```
      X1=0.
      X2=3.141592
      N=100

      H=(X2-X1)/REAL(N)

      RI=0.
      F2=F(X1)

      DO 100 I=1,N
      F1=F2
      F2=F(X1+REAL(I)*H)
      RI=RI+0.5*(F1+F2)
  100 CONTINUE

      RI=H*RI
      WRITE(*,*) RI

      STOP
      END

      FUNCTION F(X)

      F=SIN(X)

      RETURN
      END
```

그림 1. 정적분 Fortran program

```
do(0) :- !.
do(I) :- J=I-1,
         do(J),
         f2(F2),retract(f1(_)),
         assertz(f1(F2)),
         x1(X1),h(H),X=X1+I*H,f(NF2,X),
         retract(f2(_)),
         assertz(f2(NF2)),
         ri(RI),f1(F1),f2(NF2),
         NRI=RI+0.5*(F1+NF2),
         retract(ri(_)),
         assertz(ri(NRI)).

step4 :- ri(RI),h(H),NRI=RI*H,
         retract(ri(_)),
         assertz(ri(NRI)).

f(F,X):- F=sin(X).

goal
  reset,fail;
  step1,
  step2,
  step3,
  do(100),
  step4,
  ri(RI),write(RI).
```

그림 2. 정적분 prolog program

```
database
  x1(real)
  x2(real)
  n(integer)
  h(real)
  ri(real)
  f1(real)
  f2(real)

predicates
  reset
  step1
  step2
  step3
  do(integer)
  step4
  f(real,real)

clauses
  reset :- retract(x1(_)),fail;
           retract(x2(_)),fail;
           retract(n(_)),fail;
           retract(ri(_)),fail;
           retract(h(_)),fail;
           retract(f1(_)),fail;
           retract(f2(_)),fail.

  step1 :- assertz(x1(0.0)),
           assertz(x2(3.141592)),
           assertz(n(100)).
  step2 :- x1(X1),x2(X2),n(N),
           H=(X2-X1)/N,assertz(h(H)).
  step3 :- assertz(ri(0.0)),x1(X1),
           f(F,X1),assertz(f2(F)),
           assertz(f1(0.0)).
```

prolog는 array형의 데이타 구조가 없다. 그러나 database를 적절히 이용하면 벡터나 행렬도 prolog내에서 꽤 효과적으로 다루어 질 수 있다. 그림 3 에 그 예를 보였다. 그림에서 name은 벡터나 행렬의 이름이 들어가는 곳이며 reallist에 벡터나 행렬의 요소들이 기록된다. matrix에서 index는 행렬의 열을 나타낸다.

```
domains
  reallist=real*
  name=string
  index=integer

database
  vector(name,reallist)
  matrix(name,index,reallist)
```

그림 3. 벡터와 행렬의 구조

이러한 벡터및 행렬의 구조를 바탕으로 데이타의 처리를 효과적으로 수행하기 위해서는 벡터및 행렬의 각 연산을 행하는 predicate가 개발되어야 한다. 이들은 한번 개발된 이후에는 subroutine이나 function처럼 편리하게 사용 될 수 있다. 그림 4 와 5 에 개발된 routine의 예로서 벡터 덧셈과 행렬 곱셈을 수행하는 routine을 제시하였다. 100x100 행렬로 이들 routine의 효율을 측정한 결과 Fortran과 비슷한 수준의 실행속도를 보였다.

```
domains
  rlist=real*

database
  vec(string,rlist)

predicates
  add(string,string,string)
  a(rlist,rlist,rlist)

clauses

  add(C,A,B):- vec(A,AL),vec(B,BL),
               a(CL,AL,BL),
  _            assertz(vec(C,CL)).

  a([],[],[]):-!.
  a([Hc|Tc],[Ha|Ta],[Hb|Tb]):-
                    a(Tc,Ta,Tb),Hc=Ha+Hb.

goal
  consult("data1"),fail;
  add("c","a","b"),fail;
  save("out1").
```

(a) program

```
vec("a",[ 1, 3, 4])
vec("b",[ 3,-4, 2])
```

(b) input datafile data1

```
vec("a",[1,3,4])
vec("b",[3,-4,2])
vec("c",[4,-1,6])
```

(c) output datafile out1

그림 4. 벡터 덧셈 routine (c=a+b)

```
domains
  rlist=real*

database
  vec(string,rlist)
  mat(string,integer,rlist)

predicates
  multip(string,string,string)
  innerp(real,rlist,rlist)
  append(rlist,rlist,rlist)

clauses

  multip(B,A,X):- assertz(vec(B,[])),
                  vec(X,XL),!,
                  mat(A,_,AL),vec(B,T),
                  innerp(V,AL,XL),
                  append(T,[V],L),
                  retract(vec(B,T)),
                  assertz(vec(B,L)),
                  fail.

  append([],List,List):-!.
  append([X|L1],List2,[X|L3]):-
                    append(L1,List2,L3).

  innerp(0,[],[]):- !.
  innerp(VN,[HX|TX],[HY|TY]):-
                       innerp(V,TX,TY),
                       VN=V+HX*HY.

goal
  consult("data2"),fail;
  multip("b","A","x"),fail;
  save("out2").
```

(a) program

```
mat("A",1,[ 1, 2, 3])
mat("A",2,[-1, 2, 4])
mat("A",3,[ 2, 3,-2])
vec("x",[ 1, 3, 2])
```

(b) input datafile data2

```
vec("x",[1,3,2])
vec("b",[13,13,7])
mat("A",1,[1,2,3])
mat("A",2,[-1,2,4])
mat("A",3,[2,3,-2])
```

(c) output datafile out2

그림 5. 행렬 곱셈 routine (b=Ax)

2. 다른 언어와의 연결

IBM PC 에서 작동하는 Turbo prolog 는 다른 언어와 연결 할 수 있는 길이 열려있다. 이는 prolog의 predicate를 다른언어로 구성하는 것이다. 이때 연결이 가능한 언어는 assembler, C 등인데 이들은 argument의 전달방법을 자유롭게 제어할 수 있는 언어 이기때문이다. 이들 이외의 언어도 이같은 문제를 해결할 수 있다면 연결이 가능하다.

- C 와의 연결 -

다음의 예는 두개의 수를 더하는 C 프로그램과의 연결 예이다.

```
global domains
  arg1=integer
  arg2=integer
  arg3=integer

global predicates
  cpinit language c
  _addint(arg1,arg2,arg3)-(i,i,o) language c

goal
  cpinit,
  clearwindow,
  A=5,
  B=3,
  _addint(A,B,C),
  write(C),nl.

addint_0(a,b,c)
int a,b,*c;
{
 *c=a+b;
 }
```

그림 6. C로 기술된 prolog global predicate

prolog의 predicate addint는 세개의 argument를 가지고 있는데 그 flow pattern이 (i,i,o)로 선언되었다. 즉 첫번째, 두번째 argument는 입력이며 세번째 argument는 출력이다. 이 때 다른 flow pattern으로 사용하고 싶다면, 예를 들어 세개의 argument가 모두 입력인 flow pattern도 이용하고 싶다면 flow pattern을 다음과 같이 한번 더 선언하여야 한다.

```
_addint(arg1,arg2,arg3)-(i,i,o),(i,i,i) language c
```

flow pattern이 (i,i,o) 일 때 stack의 구조는 다음과 같다.

[BP]+10	argument3의 값이 저장되는 곳의 주소 size = 4 bytes
[BP]+8	argument2의 값 size = 2 bytes
[BP]+6	argument1의 값 size = 2 bytes
[BP]+2	addint가 수행된 후 수행되어야 하는 곳의 주소 size = 4 bytes
[BP]+0	addint가 수행되기 전의 BP size = 2 bytes

그림 7. stack의 구조

prolog에서 argument가 전달되는 방식은 입력 argument의 경우는 값(call by value)이며, 출력 argument의 경우는 주소(call by reference)이며 나중에 나오는 argument부터 stack에 저장된다. 즉 argument3의 주소가 첫번째, argument2가 두번째, argument1이 가장 늦게 stack으로 옮겨지며(PUSH) 호출된 predicate가 수행된 후 return되는 곳의 주소가 옮겨지면 stack이 완성된다.

입력 argument는 값이 전달되며 출력 argument는 주소가 전달되므로 flow pattern이 달라지면 stack의 구조가 달라지며 이에 따라 C subroutine도 달라져야 한다. 즉 각각의 flow pattern마다 그에 적합한 argument 전달방식을 갖는 subroutine이 연결되어야 한다. 그러므로 같은 이름의 prolog predicate에 flow pattern 갯수 만큼의 subroutine이 필요하며 이때 C Subroutine들의 이름은 다음의 예와 같이 정해진다.

Prolog predicate addint 앞에 붙은 "_"(under bar)는 C function이 compile 되는 과정에서 compiler가 C의 function name 앞에 붙여 주므로 prolog에서 도 붙여주어 _addint와 같이 만들어 준다. 실제로 linker는 _addint_0()와 _addint_1()라는 이름을 찾아 연결시키게 된다.

```
_addint(arg1,arg2,arg3)-(i,i,o),(i,o,i) language c


addint_0(a,b,c)
int a,b,*c;
{
 *c=a+b;
 }

addint_1(a,b,c)
int a,*b,c;
{
 *b=a+c;
 }
```

그림 8. flow pattern에 따른 C subroutine들

prolog가 predicate (다른 언어에서의 subroutine, function)를 call할 때는 predicate의 주소를 segment, offset 모두 지정하는 far call을 행한다. 그러므로 prolog에 연결되는 subroutine도 far call이 가능한 memory model 로 compile이 되어야 한다. Turbo C에는 여섯가지의 memory model이 있는데 이중 large와 Huge 2개의 momory model이 prolog와 연결가능한 memory model이다.

Prolog의 프로그램이 example.pro이며 C의 부프로그램이 add.c이고 prolog의 library가 prolog.lib, C의 Library가 clib.lib이라면 이들의 연결은 다음과 같이 행해져야 한다.

```
link
init+cpinit+example+add+example.sym,mixproc,,prolog+clib
```

이때 init의 file이름은 init.doj이며 prolog 프로그램의 초기화 부프로그램 이며 cpinit는 cpinit.obj이며 Prolog와 C를 연결할 때 필요한 file이고 example.sym은 prolog 프로그램을 compile할 때 만들어지는 file이다. 연결할 때 주의해야 할점은 example.sym이 반드시 연결되는 file 중 마지막 에 위치해야 하는 것이다.

결 론

1. Prolog로 Fortran과 유사한 효율을 갖는 수치 data 처리 routine이개발될 수 있었다.
2. Prolog로 Procedual(Algorithmic)한 Program도 작성될 수 있었다.
3. Prolog와 다른 language를 연결하여 사용할 수 있었다.
4. 위의 기법을 전력계통의 전문가 시스템에 적용할 예정이다.

참고 문헌

[1] "Turbo prolog", Borland, 1986

[2] "Turbo C reference guide", Borland, 1987

[3] "Turbo C user's guide", Borland, 1987

[4] 황희용, "C 언어기초 + @ ", 교학사, 1987

[5] Ivan Bratko, "PROLOG programming for artificial intelligence", Addison - Wesley, 1986

[6] Eugence Charniak, Drew McDermott, "Introduction to Artificial Intelligence", Addison - Wesley, 1985

[7] Paul Harmon and David King, "Expert Systems", John Wiley & Sons, 1985