벡트란 코드화를 위한 프로그램 변환과 단순화

황 선 명† 김 행 곤††

요 약

기존 포트란 프로그램을 벡터 처리 가능한 코드인 벡트란으로 변환시키는데 있어서 가장 문제가 되는 것은 루프내에서 제어의 흐름가 발생하는 경우 조건적 전달이 일어난다는 것이다. 조건적 전달이란 어떠 문장의 실행이 다른 문장내의 변수 값에 의해 이루어지는 제어 의존성으로, 본 논문은 루프내부의 조건적 제어를 제거하기 위한 알고리즘과 조건적 할당문을 이용하였을때 그 내부의 복잡한 조건에 대한 단순화 알고리즘을 제시한다. 이 때 조건적 할당문의 조건은 부울 변수 (2-상태)뿐아니라 3가지 이상의 상태를 나타내는 n-상태 변수를 통하여 나타낸다.

On the Program Conversion and Conditional Simplification for VECTRAN Code

Sun Myung HWANG† and Haeng Kon KIM††

ABSTRACT

One of the most common problems encountered in the automatic translation of FORTRAN source code to VECTRAN is the occurence of conditional transfer of control within loops. Transfers of control create control dependencies, in which the execution of a statement is dependent on the value of a variable in another statement.

In this paper I propose algorithms involve an attempt to convert statements in the loop into conditional assignment statements that can be easily analyzed for data dependency, and this paper presents a simplification method for conditional assignment statement. Especially, I propose not only a method for simplifying boolean functions but extended method for n-state functions.

1. 서 론

복잡한 프로그램 제어 흐름에 의한 의존성(dependence)을 자료흐름에 관한 의존성으로 변환시키는 동시에 이를 벡터 연산에 적합한 형태의 프로그램으로 변환하기 위해서 새로운 번역기가 필요하다. 즉 자동 벡터화를 위한 프로그램 분석방법들은 문장들간의 의존성을 기본개념으로 하고 있으며 순차 프로그램(sequential program)을 벡터화 또는 병렬처리 가능한 코드로 만드는 것이 필요하다.

ANSI에서 제정하게 될 차기 포트란 언어는 이러한 벡터 연산을 갖게될 것이 명백하며 이를 통하여 프로그래머들은 벡터컴퓨터를 효율적으로 이용하여 프로그램의 효율성과 재사용성을 높일 수 있게 될 것이다. 현존하는 포트란 코드를 벡터컴퓨터으로 이용가능하게 하려면 현존하는 코드를 벡터연산 가능코드로 변환시켜야 하며 이러한 프로그램상의 변환위치는 주로 DO 루프지점이 된다. 현재 미국 Rice 대학에서는 PFC(Parallel Fortran Converter)라고 알려진 번역기를 개발중에 있으며 KUCK 등이 연구한 최적화 컴파일-프리프로세서 PARA FRASE와 독일 Bonn대학의 INFORMATIK에서 수행중인 SUPRENUM프로

· 이 논문은 1993년도 한국 학술진흥재단의 지방대 조성 학술 연구조성비에 의하여 연구되었음.

† 정 회 원 : 대전대학교 전자계산학과 조교수
†† 정 회 원 : 효성여자대학교 전자계산학과 조교수

논문접수 : 1994년 3월 3일, 심사완료 : 1994년 5월 18일

젝트 연구의 핵심도 바로 DO루프내에서 벡터연산 가능코드로의 변환과정이다[1].

기존 포트란 코드를 벡터 처리가 가능한 포트란(벡트란) 코드로 자동 변화하는데 있어서 가장 큰 문제는 루프(loop) 안에서 제어의 조건적 전달들(control transfers)이다[1].

조건적 전달이란 어떤 문장의 실행이 다른 문장 내의 변수 값에 의해 이루어지는 제어 의존성(control dependency)을 만드는 것을 말한다. 자동변환기는 병렬 처리가 되도록 벡터 문장들을 만드는 시스템이며 이를 방해하는 문장 즉 산술 IF문, 논리 IF문, GO TO문, 할당형 GO TO문에 의해 발생되는 의존성을 해결할 때 비로소 가능하다. 이를 위한 보다 명확한 해법은 제어 전달들을 제거시키는 것이고 보다 확실한 자료의 의존성을 명시하는 방법으로 문장들의 실행 조건을 조절해야 한다[5][6].

내부 루프로부터 제어의 조건 전달을 제거하기 위하여 트랜스레이터에서 사용하는 두가지 방법은 다음과 같다.

1) IF 복사 (copying)
2) 조건 할당문(conditional assignment statement)

IF 복사는 논리 IF에 의해 검사되는 조건이 루프 불변성(loop invariant)일 때 사용하며 루프 스위칭(loop switching)과 같이 컴파일러의 최적화에 친숙한 기법이다.

두번째 방법은 루프내의 문장들을 자료 의존성(data dependency)에 대한 분석이 쉬운 조건 할당문으로 바꾸는 기능을 가진다. 조건 할당문들은 독립성의 기준을 만족할때 조건적 벡터 할당문들로 쉽게 변환시킬 수 있다[5]. 본 논문은 DO루프내에 존재하는 문장들에 대하여 의존성 분석을 통한 IF 변환 알고리즘과 조건할당문을 이용하여 조건 전달을 해결할 때 이 과정에서 발생하는 조건의 단순화 과정을 제시한다.

2. IF 변환

2.1 IF 변환의 세 요소

벡터 연산이 가능하도록 기존문장을 벡터화 시키는 방식에는 두가지가 있다[1][7][8].

(1) 컴파일러에 의한 벡터화 : 컴파일러가 기존의 프로그램을 효율적인 벡터 코드로 변환할 수 없기 때문에 프로그래머가 다시 자신의 프로그램을 수정해서 컴파일해야 하는 번거러움이 있다.
(2) 벡터 번역기에 의한 벡터화 : 기존 포트란 프로그램을 먼저 번역기를 통해 한번만 번역하여 컴파일 시키는 방법으로 프로그램을 재작성할 필요가 없어진다.

본 논문에서는 번역기를 통한 벡터화를 목적으로 병행성이 가장 많은 루프 내부의 문장에 대하여 벡터연산이 가능토록 변환하는 과정과 알고리즘을 제시한다.

먼저 포트란 명령의 IF 변환 대상이 되는 4가지 문장을 살펴보자[6].

(1) 할당문
(2) 분기(branch)
(3) 반복문
(4) 위치지정문

위의 분류에 IF문이 없는 이유는 임의의 할당문이나 분기문에 첨가되어질 수 있는 요소로서 IF를 다루기 때문이다. 즉 모든 할당문이나 분기문은 조건문으로 생각할 수 있다.

본 논문에서 설명하는 IF 변환과 단순화 방법은 주어진 프로그램내에 존재하는 모든 goto문들을 제거시키는 방법과 알고리즘으로 그 대상은 분기문으로 하였으며 원시프로그램의 실행순서는 각 할당문에 대한 논리조건을 계산함으로써 변경되지 않으며 그대로 유지된다. 이때 이러한 논리조건을 가드(guard)라도 정의한다.

가드는 그 문장의 실행가능 조건을 부울 표현으로 정의한 것이다. 즉, 정의된 부울 표현이 '참(true)'일때 해당 문장이 실행된다.

원시프로그램을 분석하기 위하여 분기의 형태를

나누어 보면 다음과 같다.

(1 탈출 분기(exit branch) : 하나 이상의 루프를 종료하는 분기이다.

```
      DO 20 I=1,100
        IF (A(I)-B(I).GT.C(I)) GO TO 30
        . . .
20    continue
        . . .
30    continue
```

(2 전방 분기(forward branch) : 분기의 목표가 되는 문장이 그 분기 뒤에 존재함과 동시에 동일한 레벨의 루프안에 존재하는 분기이다.

```
      DO 10 I=1,100
        IF (A(I).EQ.0) GO TO 20
        B(I)=B(I).A(I)
20    continue
10    continue
```

(3 후방 분기(backward branch) : 분기가 발생하기 이전에 분기의 목표가 존재한다.

```
10  I=I+1
    A(I)=A(I)+B(I)
    IF (I.LE.100) GO TO 10
```

2.2 탈출 분기와 분기의 재배치(relocation)

탈출 분기는 그 분기 발생 전과 발생 후에 있는 문장들의 실행에 모두 영향을 미치게 되므로 다른 분기와 차이가 있다. 다시 말하면, DO루프를 탈출하는 분기에 의해 루프가 더 이상 실행되지 않는다면 그것은 루프내의 모든 문장들에게 영향을 주게 된다.

```
    DO 100 I=1,100
      S1
      IF (X(I)) GO TO 200
      S2
100 continue
    S3
200 S4
```

위의 예에서 분기가 발생되면 DO 루프는 실행을 마치게 되어 S1,S2 중 어느것도 더 이상 실행될 수 없다. 만일 위의 루프 대신에 다음과 같은 단순 전방 분기일 경우

```
      S1
      IF (X(I)) GOTO 200
      S2
      S3
200   S4
```

S1은 분기와는 전혀 무관하게 실행되어진다. 그러므로 탈출 분기는 전방 분기보다 복잡한 제어 의존성을 갖게 되며 이들을 제거하기 위하여 루프내의 모든 문장들의 가드들을 수정해야만 한다. 만일 모든 탈출 분기들이 전방이나 후방 분기로 전환시킬 수 있다면 IF변환의 문제는 보다 간단해지게 된다. 즉 IF 변환 알고리즘을 통해 모든 분기들이 DO 루프의 동일 단계에서만 점프하도록 분기를 재배치하면 전방 분기들은 분기 제거단계에서 제거가 수월해 진다.

분기 재배치와 다음 단계에서 설명될 분기제거의 기본과정은 각 문장에 대한 부울 가드 수식을 구하는 것이다. 루프 밖으로 탈출 분기를 이동시키는 재배치 과정에서 나타나는 루프내의 각 문장들의 가드들은 탈출 플래그(flag)의 논리곱(conjunction)에 의해 완전한 가드로 완성된다. 이 때 완성된 가드의 요소들에 적용될 연산들은 논리곱과 논리합(disjunction), 그리고 부정(negation)이다.

탈출 플래그를 실제로 계산하기 위하여 이와 대응하는 논리변수 EXi를 정의하고 μ(exi)=EXi인 루프 탈출 조건을 만족하도록 한다. 만일 루프안에 다음의 형태인 탈출분기가 존재한다면

```
IF (P) GOTO S1
```

이때의 탈출분기가 발생하지 않는 가장 최근 실행 조건인

```
EXi=.NOT. P
```

로 대체되어 새로운 분기 형태인

```
IF(.NOT.EXi) GOTO S1
```

이 생성되어 루프의 바로 다음에 배치된다. 이때의 분기는 이미 루프를 벗어났으므로 탈출 분기가 아닌 전방 또는 후방 분기로서 가장 복잡한 탈출 분기문제가 해결되어진다. 루프안에 있는 새롭게 생성된 할당문을 포함한 모든 문장들의 가드들은 그 루프에 대한 각 탈출 플러그에 의해 다음과 같이 구성된다.

$$EX1 \wedge EX2 \wedge EX3 \cdots \wedge EXn$$

달에서의 예에 대해서 재배치 후 상태를 보면 다음과 같다.

```
      EX1= .TURE.
      DO 100 I=1, 100
         IF (EX1) S1
         IF (EX1) EX1= .NOT. X(1)
         IF (EX1) S2
100   CONTINUE
      IF (.NOT. EX1) GOTO 200
      S3
200   S4
```

2.3 전방 분기와 제거 알고리즘

제어종속의 가장 간단한 형태는 전방 분기이다. 그 분기와 목표사이에 있는 문장들의 실행은 분기 수식(brach expression)에 있는 변수들의 값에 좌우되기 때문에 IF변환은 이 종속을 정확히 반영하는 가드들을 결정하여야 하며 이 가드들로 전방 분기는 제거된다.

분기제거(branch removal)의 모든 과정에 기본이 되는 것은 현재 조건이며 이는 현재 주어진 문장의 실행조건들을 논리적으로 나타낸 것이다. 분기제거가 프로그램내의 문장에서 다른 문장으로 옮겨감에 따라서 다음 문장에 대한 가드를 생성하기 위한 현재조건을 가진 부울 변수를 결합 또는 분리하게 되는데 바로 이 부울변수들이 프로그램의 전방 분기들에 관한 사항들을 나타낸다.

전방 분기는 두개의 위치에서 제어흐름에 영향을 준다. 즉 제어흐름이 순차적인 흐름에서 부터 이탈될 수 있는 분기점과 분할된 흐름들이 재결합하는 목표점에서이다. 그러므로 이들 지점에서 현재조건(CC)는 전방 분기를 제거하기 위하여 수정되어야 한다.

(1) 분기점에서의 CC

전방 분기 바로 다음 문장은 제어흐름이 분기에까지 도달한 후 그 분기조건을 만족하지 않을 때만 실행될 수 있다. 그러므로 전방 분기에서 현재조건이 CC1이고 분기의 프래디킷(predicate)가 P라면 분기 다음 문장에 대한 가드는 CC1· P가 된다.

(2) 목표에서의 CC

목표에 도착되는 제어 흐름은 바로 이전 문장에서부터 순차적으로 도착되든지 분기 자체를 통하는 것이다. 만일 목표 이전 문장의 가드가 CC2라고 하면, 목표의 가드는 CC2∨(CC1 ∧P)이 된다. 이때 분기점과 목표 사이에 제어흐름의 변화가 없다면 CC2=CC1· P이며, 목표 문장의 가드는 (CC1 ∧¬∨(CC1 ∧P)이 되어 이는 CC1으로 단순화된다.

이 같은 내용을 다음의 예를 통하여 살펴보기로 한다.

```
          DO 100 I=1,100
             IF (A(I).GT.10) GOTO 60
S1                 A(I)=A(I)+10
                   IF(B(I).GT.10) GOTO 80
S2                      B(I)=B(I)+10
S3    60           A(I)=B(I)+10
S4    80     B(I)=A(I)-5
      100 CONTINUE
```

루프안의 두 분기조건을 갖는 부울 변수 br1과 br2를 정의한다. 이러한 분기 플래그에 대응하여 포트란 논리변수 BR1과 BR2를 BR1=μ(br1), br2=μ(br2)로 이용하는데 위 프로그램 텍스트에서 이들은 IF문 대신에 다음의 치환문을 삽입함으로 완성되어진다.

BR1=A(I).GT.10
BR2=B(I).GT.10

위에서 설명된 전방 분기 제거 방법에 따르면 루프안의 문장들은 다음의 조건들에 의해 제어된다는 것을 알 수 있다.

문 장	제 어 조 건
S1	¬ br1
S2	¬ br1∧¬ br2
S8	br1∨(¬ br1∧¬ br2)
S4	br1∨(¬ br1∧¬ br2)∨ ¬ br1∧¬ br2)

BR1과 BR2같은 논리변수를 포함한 수식이 확장되고 복잡해지는 것을 방지하기 위하여 IF변환기는 논리식을 단순화하는 과정이 필수적이다. 예를 들면, S7문에 대한 제어조건은 복잡한 식으로

되는 일지만 항상 '참'이 되어야 함을 알 수 있다. 그러므로 IF 변환 후의 단순화 작업은 위 예제 프로그램을 다음과 같이 변화 시킨다.

```
        DO 100 I=1,100
           BR1=A(I).GT.10
S1         IF (.NOT.BR1)  A(I)=A(I)+10
           IF (.NOT.BR1)  BR2=B(I).GT.10
S3         IF (.NOT.BR1.AND..NOT.BR2) B(I)=B(I)+10
S4         IF (BR1.OR..NOT.BR2)    A(I)=B(I)+A(I)
S5         B(I)=A(I)-5
   100  CONTINUE
```

S3을 제어하는 조건은 예상된 것과는 차이가 있다. 이는 S4가 항상 '참'인 조건으로 단순화되었듯이 S3도 단순화 과정을 통하여 생성된 조건이다.

2.4 후방 분기와 완전 분기 제거 알고리즘

전방 분기의 제거는 매우 쉽게 이루어 질 수 있지만 제어 종속의 마지막 형태인 후방 분기는 프로그램에서 직접적으로 제거될 수 없다. 그 이유는 후방 분기는 가드화 된 문장으로 처리할 수 없는 루프 구조를 생성하기 때문이다. 그러므로 후방 분기가 존재하는 경우 아래와 같은 코드의 전방 분기 제거는 앞의 알고리즘을 적용할 수 없다.

```
        IF (X) GO TO 200
        . . .
100     S1
        . . .
200     S2
        . . .
        IF (Y) GO TO 100
```

전방 분기를 제거하기 위해 분기 제거 알고리즘을 적용하면 S1에 대한 가드는 ¬X가 되는데 이는 X가 '참'이고 Y가 '참'인 가드를 만족하지 못하므로 S1의 가드로 옳지 않다. 이와 같은 후방 분기들로 루프의 문제를 피하기 위한 IF 변환의 한가지 가능한 방법은 그들을 고립시켜서 그들의 제어하에 있는 코드 즉 후방 분기에 의한 잠정적으로 반복되어지는 영역은 건드리지 않는 것이다.

S1에 대한 가드는 다음 두가지 경우를 반영하여야만 한다.

(1) S1은 X가 '거짓'일때만 첫번째 경로에서 실행되며

(2) S1은 후방 분기를 취하는 경우에만 항상 실행된다.

즉 하나의 가드 조건은 첫번째 경로에서는 '거짓'인 항목과 후방 분기를 취했을 때 '참'인 부울 변수를 사용하여 결정할 수 있다. 이때 후방 점프를 취했음을 후방 분기 프래그 bb로서 명시한다.

이상을 앞의 예제에 적용하면 다음과 같다.

```
        문 장                  제 어 조 건
        BR1=X                  ture
        . . .                  ¬ br1
        BB1=.FALSE.            ture
100     S1                     ¬ br1∨(br1 ^ bb1)
        . . .                  ¬ br1∨(br1 ^ bb1)
200     S2                     ture
        .
        IF (Y) THEN
           BB1=.TURE.
           GO TO 100
        ENDIF
```

후방 분기의 목표에서의 가드는

$$CC_Y \vee (br1 \wedge bb)$$

가 되며 만일 반복지역으로 한 개 이상의 점프가 존재한다면 두번째 항목은 bbj와 결합된 각 분기조건의 논리합이어야 한다.

```
        IF (X) GO TO 20
100     S1
        GO TO 300
200     S2
        IF (Y) GO TO 100
300     S3
```

이 예제에서 S2에 대한 올바른 가드는 200으로의 전방 분기가 발생하고 후방 분기가 발생하지 않아야 하므로 br1 ^bb1이 되어야 한다. 이때 S2 이전의 분기를 제거하기 위하여 bb1 항이 있어야 한다.

```
        BR1=X
100     IF (.NOT. BR1 .OR. BB1 .AND. BR1) S1
        /* Go To 300 has been eliminated          */
200     IF (.NOT. BB1 .AND. BR1) S2
        IF (.NOT. BB1 .AND. BR1 .AND. Y) THEN
           BB1=.TURE.
           GO TO  100
        ENDIF
300     S3
```

3.부울 함수의 단순화

3.1 조건할당문

앞장에서 제시한 IF변환과 각 알고리즘에 의하여 DO문장내의 할당문이 변화됨을 보았다. 이러한 IF변환은 각 할당문의 실행 여부를 결정하는 가드에 의해 그 조건이 만들어지는데 이를 단순화하는 작업이 뒤따르게 된다. 앞 장에서 보았듯이 IF변환시의 변환과정을 실행하는 조건 할당문의 형태는 다음과 같다.

IF (condition) assignment

그러므로 다음과 같은 코드로 주어졌을 때

```
      DO 10 I = 1,100
         IF (X(I) .GT. 10  GOTO 10
           X(I) = function1 (X(I))
           Y(I) = function2 (X(I))
10    CONTINUE
```

포트란 소스 코드에서 요구되는 변환은 다음과 같다.

```
DO 10 I = 1,100
  IF (.NOT. (X(I) .GT. 10)) X(I) = function1(X(I))
  IF (.NOT. (X(I) .GT. 10.) Y(I) = function2(X(I))
10  CONTINUE
```

또한 Range 문장안에 명시된 배열의 범위에 관한 오퍼레이션을 '*'로 표시할때 의존성 분석(dependency analysis)이 끝난 뒤에는 다음과 같은 벡트란 결과가 나타난다. 이때 WHERE 다음의 조건에 따라서 다음 할당문의 실행이 결정된다.

```
RANGE /M/X, Y
M = 100
WHERE (.NOT. (X(*) .GT. 10 ) X(*)=function1 (X(*))
WHERE (.NOT. (X(*) .GT. 10 ) Y(*)=function2 (X(*))
```

그러나 모든 변형된 코드가 위의 예와 같이 단순하지는 않다. 많은 경우 한 문장이 여러개의 조건적 GOTO 또는 IF의 분기의 영향에 따라 실행이 결정된다.

```
   DO 100 I = 1, 100
    C1 = A(I) + 3
 20 IF (C1 .LT. 0) A(I) = -A(I)
    C2 = B(I) .GT. 10
 30 IF ((C1 .LT. 0) .AND. (.NOT. C2)) B(I)=B(I)-10
 40 IF (((C1 .LT. 0) .AND. (.NOT.C2)).OR. C1.EQ.0
        A(I) = A(I) + B(I)
 50 IF (((C1.LT.0).AND.(.NOT.C2)).OR.(C1.EQ.0 .OR.
      ((C1 .LT. 0) .AND. C2)) B(I) = A(I) + B(I)
 60 IF (((C1.LT.0).AND.(.NOT.C2)).OR.(C1.EQ.0)).OR.
      ((C1. LT. 0) .AND. C2) .OR. (C1 .GT. 0))
         C(I) = A(I) + B(I)
100 CONTINUE
```

이와 같이 여러 분기들이 재결합된 상태로의 조건들은 그 구성이 매우 복잡함으로 이를 그대로 처리하기에는 많은 과부하가 발생하게 된다. 그러므로 IF변환시 발생되는 조건의 단순화된 형태를 만드는 방법이 필요하며 다음절과 같은 제안에 따라 예제의 50, 60번 문장을 단순화 할 경우 다음과 같다.

```
   DO 100 I = 1, 100
    C1 = A(I) - 3
 20 IF (C1 .LT. 0) A(I) = -A(I)
    C2 = B(I) .GT. 10
 30 IF ((C1 .LT. 0) .AND. (.NOT. C2)) B(I)=B(I)-10
 40 IF (((C1 .LT. 0).AND.(.NOT.C2)).OR.(C1.EQ.0)))
        A(I) = A(I) + B(I)
 50 IF (C1 .LE. 0) B(I) = A(I) + B(I)
 60 C(I) = A(I) + B(I)
100 CONTINUE
```

벡트란에서 사용하는 조건 단순화는 k개 항(term)들의 합으로 구별된 하나의 조건을 받아들인다. 이때 항들은 중간항(minterm)들의 집합을 생산하기 위해 확장되고 중간항이 재조합되어 다음의 특성들을 만족하도록 조건의 최소 적의합(sum of product) 표현을 얻기 위한 항들이 선택된다.

1) 각 항의 변수들의 수가 최소화
2) 조건내 항들의 수를 최소화

이 결과 조건 단순화 과정의 출력은 새로운 최소화 표현이 된다.

3.2 부울 함수의 단순화

부울함수에 대한 단순화 방법은 먼저 상위 수준의 항들에서 모든 가능한 중간항들의 조합을 찾아낸후 그 함수를 실현시키기 위한 항들의 최소 조합을 선택하는 매우 명확한 과정이다[2][3].

이 과정의 입력은 함수를 구성하는 항들의 집합이다. 이러한 항들은 중간항들로 확장되고, 결국

중간항들의 집합에 의해 함수는 구체화된다. 중간항들 또는 0-cube들로의 확장은 다음 단계 과정에서 조합될 때 필요하다. 중간항으로의 표현은 각 변수들에 대한 하나의 상태로 명시하고 조합들을 만들기 전에 필요한 항들의 비교를 쉽게 한다.

각 중간항은 다른 모든 중간항과 비교되는데 만일 두 중간항이 모든 변수의 위치에서 동일하다면 그들은 1-cube를 형성하도록 결합된다. 1-cube는 0-cube 또는 중간항의 상위 레벨이며 "don't care"는 0-cube에 의해 표시되는 "참"과 "거짓"을 둘다 함축한다. 만일 두 개의 0-cube가 합하여진다면 그들의 상위 레벨에 의해 커버(cover)된다는 것을 의미한다. 이러한 정보는 후에 함수의 기본함축체(prime implicant)의 집합을 선택할때 사용된다.

모든 가능한 1-cube들이 생성되어지면 각 1-cube는 다른 모든 1-cube와 비교되며 이때 두 1-cube가 모든 변수의 위치에서 동일하다면 그들은 2-cube를 형성하도록 합하여 지며 이때의 1-cube들은 커버된 것으로 표시된다. 같은 방법에 의하여 i-cube들은 더이상 조합이 만들어지지 않을때까지 (i+1)-cube를 생성하기 위해 결합된다.

이 과정이 끝나면 초기상태의 항들로 부터 생성될 수 있는 모든 가능한 항들의 집합이 만들어진다.

기본함축체의 집합은 가능한 모든 cube들의 집합으로 부터 상위 레벨 cube 리스트에 의해 커버되는 cube들을 제거시킴으로 얻어지는데 이러한 기본함축체들의 집합은 원래 함수를 만족하는 최소 최적 항들의 집합으로 표현되어져야 한다.

다음 단계는 기본함축체들로 부터 필수함축체(essential implicant)들을 찾는 것이다. 이를 위하여 기본함축체들과 이들이 커버하는 함수의 항들간의 관계를 표시하는 표가 만들어진다. 만일 필수함축체가 어떤 중간항을 커버하는 유일한 함축체라면 그것은 필수함축체로 간주한다. 모든 항들이 필수함축체들에 의해 커버되어졌을때 그 결과는 평가되어 진다.

만일 함수의 임의의 중간항이 커버되지 않는다면 기본함축체들의 축소된 표는 필수함축체들에 의해 커버되지 않는 중간항들만을 고려하여 만들고 그 결과 표에서 제거되지 않는 중간항들을 커버하는 2차의 기본함축체들을 표로부터 찾아야 한다. 또한 이 표는 다른 항에의해 지배되어 지거나 상호 교체 가능한 항이 존재할때 표상에 있는 항들은 제거되어 질 수 있다. 상호 교체 가능한 항이라는 것은 두 항이 동일한 중간항을 커버하고 변수들의 같은 수에 의하여 표현될 때를 말한다. 한 항이 다른 항을 지배한다는 것은 그것이 동일한 중간항들을 커버하고 보다 적은 변수들로써 표현되거나, 그것이 동일한 중간항들과 하나 이상의 다른 중간항들을 커버하고 동일한 수 또는 적은 변수들로 표현되어 질 때를 말한다. 이러한 축소표를 만들때는 이전에 커버되지 않은 중간항들 만이 대상이 된다. 축소된 표를 이용하여 2차의 필수함축체들을 선택하는데 모든 중간항들이 커버될때까지 이 과정을 반복한다.

만일 모든 중간항들이 커버되기 이전에 중간항들이 존재하지 않는 경우가 있는데 이것은 하나의 기본함축체가 순환적으로 존재하기 때문이다. 이러한 순환은 어떤 기본함축체를 선택하고 그것을 커버하는 중간항들을 만드는 것에 의하여 멈추어 진다. 필수함축체로서 표시된 것들은 함수의 최소 적의합 표현으로 구성되며 이것을 포트란의 IF 변환문에서 조건으로 사용하게 된다.

3.3 단순화 과정의 예

이러한 전반적인 과정을 예제로 통해서 살펴보면 다음의 함수가 IF의 조건으로 주어진다고 하자.

$$\overline{AB}\,\overline{CD} + \overline{A}\,\overline{BC} + \overline{AB}\,\overline{C} + A\overline{BCD} + \overline{A}BC + \overline{ABC}D$$

이 함수를 중간항 표현으로 나누어서 다루기로 한다.

$$\overline{AB}\,\overline{CD} + (\overline{ABC} + \overline{ABC}D) + (A\overline{BCD} + \overline{ABC}D)$$

$$+ \overline{ABC}D + (\overline{ABC} + \overline{ABC}D) + \overline{ABC}D$$

항들은 하나의 열(column)로써 나타내고 열의 위치는 variable의 명칭을 결정하는데 사용되면, 변수들의 값이 참일때 "T", 거짓일때 "F", "don' care"일때 X로 표시할 수 있다.

첫번째 것은 함수로 부터 얻은 중간항들의 리스트이고, 두번째는 0-cube들의 가능한 모든 조합으로부터 얻어지는 1-cube들의 리스트이며, 세번째 것은 1-cube들의 조합에 의한 2-cube의 리스트이다. 이때, '+' 표시는 상위 레벨 항에 의해 커버되어지는 항들을 표시한 것이다.

0-cube	1-cube	2-cube
F F T F +	F F X F +	X F X F
F F F F +	X F T F +	F X F X
F F F T +	F F F X +	
F T F F +	F X F F +	
F T F T +	X F F F +	
T T T T +	F X F T +	
T F T F +	F T F X +	
T F T T +	T X T T	
T F F F +	T F T X	
	T F X F +	

2-cube X F X F가 생성되는 단계를 살펴보면 0-cube의 F F T F와 F F F F는 F F X F를 구성하는데 결합되고 0-cube의 T F T F와 T F F F는 T F X F를 만드는데 결합된다. 이렇게해서 생성된 두 cube들은 다시 X F X F를 생성하게 된다.

X F X F는 또한 1-cube의 X F T F와 X F F F로 부터 만들어질 수도 있다. 결과적으로 1-cube의 두 쌍에 의해 X F X F는 생성되고, 역으로 2-cube X F X F는 이러한 cube들을 커버하게 된다.

상위 레벨 항에 의해 커버되지 않는 항들은 함수의 기본함축체들이 된다. 앞의 예에서 기본함축체를 변수들로 나타내면 다음과 같다.

$$\bar{A}CD,\ A\bar{B}C,\ B\bar{D},\ A\bar{C}$$

기본함축체와 중간항들의 관계표는 <표 1>과 같고, 이때 기본함축체가 커버하는 중간항은 해당위치에 '+' 표시를 하였다.

줄의 경계는 하나의 항이 표시되는 변수의 수에 따라서 분리한 것이며 표의 하단에는 필수함축체로서 확정된 항에 의해서 어떤 중간항이 커버되는가를 표시하였다. 이때 필수함축체에는 "*"를 붙인다. 첫번째 필수함축체를 선별한 결과가 <표 2>이다.

<표 1> 기본 함축체와 중간항의 관계

<Table 1> prime implicants/minterms

	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}BC\bar{D}$	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}C\bar{D}$	$A\bar{B}CD$	$AB\bar{C}\bar{D}$	$AB\bar{C}D$	$ABC\bar{D}$
$B\bar{D}$	+	+					+		+
$A\bar{C}$		+	+	+	+				
$\bar{A}CD$						+		+	
ABC							+	+	

<표 2> 필수 함축체의 예

<Table 3> essential implicant (1)

	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}BC\bar{D}$	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}C\bar{D}$	$A\bar{B}CD$	$AB\bar{C}\bar{D}$	$AB\bar{C}D$	$ABC\bar{D}$
* $B\bar{D}$	+	+					+		+
$A\bar{C}$		+	+	+	+				
$\bar{A}CD$						+		+	
ABC							+	+	
	+	+					+		+

<표 3> 2차 필수 함축체의 예

<Table 3> essential implicant (2)

	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}BC\bar{D}$	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}C\bar{D}$	$A\bar{B}CD$	$AB\bar{C}\bar{D}$	$AB\bar{C}D$	$ABC\bar{D}$
* $B\bar{D}$	+	+					+		+
* $A\bar{C}$		+	+	+	+				
*$\bar{A}CD$						+		+	
ABC							+	+	
+	+	+	+	+	+	+	+	+	+

결국 모든 필수함축체를 찾은 <표 3>에서 부터 최소 적의합 표현을 얻을 수 있다.

$$\overline{ABCD} + \overline{ABC} + \overline{ABC} + ABCD + \overline{ABC} + \overline{ABCD}$$

$$= ACD + \overline{BD} + \overline{AC}$$

앞의 예에서는 기본함축체를 나타내는 표가 축소되지 않았지만, 함수를 나타내는 중간항을 1에서 15번의 레이블을 붙이고 기본함축체가 a에서 h까지로 나타내는 다음의 예를 보자

<표 4> 기본 함축체와 중간항

<Table 4> prime implicants/minterms

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a		+	+			+	+	+	+					+	+
b					+		+						+		+
c	+		+		+		+								
d	+			+											
e				+							+				
f										+	+				
g									+			+			
h											+	+			

필수함축체들을 찾은 뒤의 표는 다음과 같다.

<표 5> 필수 함축체

<Table 5> essential implicants

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
*a		+	+			+	+	+	+					+	+
*b					+		+						+		+
c	+		+		+		+								
d	+			+											
e				+							+				
*f										+	+				
g									+			+			
h											+	+			
		+	+		+	+	+	+	+	+	+		+	+	+

축소된 기본함축체 표는 필수함축체에 의해 커버되는 모든 행과 열을 제거하고 나머지 중간항들과 기본함축체들로만 구성된다.

<표 6>은 2차의 필수함축체를 구하기 위한 표로써 다른 것에 의해 지배되거나 교환 가능한 행

<표 6> 축소된 prime implicant

<Table 6> reduced prime implicants

	1	4	12
c	+		
d	+	+	
e		+	
g			+
h			+

<표 7> 2차 필수 함축체

<Table 7> secondary essential implicants

	1	4	12
c	+		
**d	+	+	
**g			+
	+	+	+

(row)을 택하여 제거하면 <표 7>과 같다. 그러므로 필수함축체 a,b,f와 2차 필수함축체 d,g로 최소화된 적의합 형태인 함수를 표현할 수 있다.

function = a + b + f + d + g

4. N-상태함수의 제안

산술 IF 내의 변수나 수식은 세가지의 관련상태, 즉 0 보다 작거나, 같거나 큰 상태를 갖을 수 있다. 앞 장의 부울변수에 의한 방법으로는 이러한 상태에 관한 정보를 쉽게 표현할 수가 없다. 이를 해결하기 위하여 세가지 구별된 상태중에 임의의 한 상태를 표시하거나 세가지 상태의 임의의 조합을 나타낼 수 있는 3-상태 변수의 사용을 제안한다. 단순화할 대상 함수에서 3-상태 변수가 발생할때의 문제는 산술 IF문에 의해 생성된 3-상태 변수와 산술형 GO TO문에 의해 생성되는 n-상태 변수들을 적절히 조절하는 방법이며 이를 위하여 앞 장의 방법을 확장시켜야 한다.

부울의 경우보다 n-상태의 경우의 변수의 표현은 보다 어려운 문제로서 부울함수일때 어떤 변수 A가 "참"이면 A, "거짓" 일때 A 로써 표현하고,

그 변수의 위치에 따라 정의할 때는 "참"을 T, "거짓"을 F 그리고 "참" 또는 "거짓"(즉 "don't care")을 X로 나타내었으나 이 방법을 n-상태를 나타내는 변수에 적용하면 각 상태마다 이를 표현할 비수치적인 문자를 정의하여야 할 것이다. 그러므로 n-상태 변수들에 대한 표현으로 각 위치에 따른 수치적인 표현인 n 비트의 스트링으로 나타내어 보자.

"참"을 '10', "거짓"을 '01' "don't care"를 "참"과 "거짓"의 조합인 '11'로 표시할 수 있다.

어떤 3-상태 변수일때도 마찬가지로 첫번째 상태를 '100', 두번째는 '010', 세번째는 '001'로, "don't care"면 '111'로 나타내게 된다.

다음과 같이 공백으로 구분된 변수들로 이루어진 항을 표시한다면,

(10 00001 1100 01 010)

위의 항은 5가지 위치 변수들로 구성되어짐을 나타내고 있다.

항들을 구성하는 중간항들에까지 확장하면 n-상태변수를 가지고 있는 함수의 중간항을 정의해야 하는데 부울항인 경우의 "don't care"는 다음과 같이 확장된다.

(11)=(10)+(01)

같은 방법으로 4-상태 "don't care"의 확장은 다음과 같다.

(1111)=(1000)+(0100)+(0010)+(0001)

이와같이 하나의 변수가 상태들의 조합을 나타낸다면 다음과 같이 "don't care" 상태가 아닌 상태 변수도 아래와 같이 확장되어야만 한다.

(011)=(010)+(001)

이와같이 함수는 중간항들의 합 또는 0-cube 들로 표현된 후에 0-cube들은 1-cube들을 형성하기 위하여 조합된다. 그러나 1-cube를 만들기 위해 조합되는 중간항들을 발견하는 일은 매우 복잡하다. 그 이유는 하나의 "don't care"를 갖는 n-상태 변수에 대하여 "don't care"를 형성하는 변수의 위치를 제외하고는 n개의 항들은 모두 동일해야 하며, "don't care"가 나타나는 변수의 위치에서는 각 항들은 각기 서로 다른 상태를 표시하여야만 하기 때문이다.

다음은 세 항들이 조합되는 예이다.

(01 001 10)+(01 010 10)+(01 100 10)
=(01 111 10)

1-cube들은 "don't care" 변수상태를 갖는 항들을 갖도록 부분적인 조합들을 만들어 가는 것에 의하여 생성되며 1-cube들을 구성하기 위하여 조합되는 항들만이 상위 cube에 의해 커버된다.

모든 가능한 1-cube들이 만들어진 수, 이 과정은 2-cube들을 만들기 위하여, 그리고 3-cube등 더 이상 상위 cube가 만들어지지 않을 때까지 반복된다. 함수의 기본함축체는 상위 cube에 의해 더 이상 커버되지 않는 cube들이다. 부울함수인 경우 원래 함수의 최소의 적의합 표현을 만들기 위하여 필요한 항들의 집합이 바로 기본 함축체들의 집합을 포함하였듯이, 동일한 방법으로 기본 함축체 테이블을 만들 수 있으며 또한 부울항인 경우에서와 같은 방법으로 최적인 필수 함축체들로 구할 수 있다.

각 항안에 변수들을 비교하지 않더라도 항들이 조합될 수 없다는 것을 알 수가 있다. 부울함수인 경우 각 항안에 "참"변수 갯수의 차이가 하나 일때만 두 항을 조합될 수 있었다.

항들이 하나의 변수만 제외하고 모든 변수의 위치에서 동일하기 때문에, 이를 제외하면 "참" 엔트리의 수나 "거짓" 엔트리의 수는 일치하게 된다. 만일 이 위치에서 하나의 항이 "참"을 갖고 다른 항은 "거짓" 엔트리를 갖게 된다면 두 항간의 "참" 엔트리 수는 하나가 더 많게된다.

만일 2-상태 변수에 대하여 "참" 엔트리의 수를 2-1 카운터에 의하여 표시하고 "거짓" 엔트리의 수는 2-2 카운터(counter)에 의하여 표시한다면, 다음과 같은 2-상태와 3-상태 변수들을 가진 항들에 대하여 얻을 수 있는 정보는 다음과 같다.

```
10 010 01 01 100          10 010 01 01 010
# 2-1 = 1                 # 2-1 = 1
# 2-2 = 2                 # 2-2 = 2
```

≠ 3-1 = 1 ≠ 3-1 = 0
≠ 3-2 = 1 ≠ 3-2 = 2
≠ 3-3 = 0 ≠ 3-3 = 0

만일 항들의 상태 카운터들이 특정 변수에 대한 카운터 만을 제외하고 모두 동일하다면 항들이 합쳐질 수 있다.

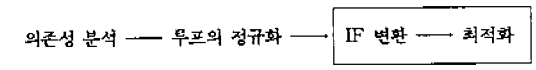
이 때 카운터 값이 다른 변수의 크기가 m이라 하면 m개의 항들이 "don't care"상태를 형성하기 위해 요구된다. 각 상태에 대한 기본 카운터(basic counter)의 값은 m개의 항들의 카운터 중 최소의 값을 선택하게 되는데, m개의 항들이 합쳐지기 위해서는 기본 카운터보다 같거나 오직 한개 큰 카운터값을 가져야만 한다. 물론 상태 카운터의 어떠한 것도 기본 카운터보다 작을 수 없다. 다음의 예는 항들과 그들의 카운터 그리고 조합을 나타내고 있다.

```
( 100 10 001 ) + ( 100 01 001 ) = ( 100 11 001 )
  #2-1 = 1          #2-1 = 0          #2-1 = 0
  #2-2 = 0          #2-2 = 1          #2-2 = 0
  #3-1 = 1          #3-1 = 1          #3-1 = 1
  #3-2 = 0          #3-2 = 0          #3-2 = 0
  #3-3 = 1          #3-3 = 1          #3-3 = 1

( 100 11 001 ) + ( 010 11 001 ) + ( 001 11 001 ) = ( 111 11 001 )
  #2-1 = 0          #2-1 = 0          #2-1 = 0          #2-1 = 0
  #2-2 = 0          #2-2 = 0          #2-2 = 0          #2-2 = 0
  #3-1 = 1          #3-1 = 0          #3-1 = 0          #3-1 = 0
  #3-2 = 0          #3-2 = 1          #3-2 = 0          #3-2 = 0
  #3-3 = 1          #3-3 = 1          #3-3 = 2          #3-3 = 1
```

5. 결 론

자동 벡터화를 도와주는 프로그램 분석 방법들은 문장들간의 의존성을 기본 개념으로 하고 있다. 본 논문은 벡트란 코드생성을 위한 프로그램 자동변환 시스템을 설계하는 과정의 일부로서



를 통하여 출력코드인 벡트란을 생성하는데, 이때 IF변환과 최적화에 관한 내용이다.

벡트란 코드를 위한 자동변환과정은 조건적 전달에 의한 방해를 받게 되는데 이러한 제어전달을 제거함과 동시에 루프내의 문장들을 적당한 조건 할당문으로 변환시켜 나가는 과정에서 유발되는 현상이 조건들의 복잡화이다. 이러한 조건들은 부울 할수로 표현되어 단순화 시킬 수 있는데 본 논문은 Quine-McCluskey 방법을 적용하여 벡트란의 조건 단순화 과정을 제시하였다. 또한 조건 할당문을 이용한 IF변환은 탈출분기, 전방분기, 후방분기에서 나타나는 상황에 따라 분기제거 알고리즘을 제시하였다.

참 고 문 헌

[1] J. R. Allen and K. Kennedy "PFC : a program to convert Fortran to Parallel form," Report MASC TR 82-6, Dept. of Mathematical Science, Rice Univ., Houston, Texas, 1982.

[2] EJ. McCluskey, "Minimization of Boolean Fuctions," Bell System Technical Journal, 35, 5, pp.1417-1444, 1956.

[3] W.V. Quine "A Way to Simplify Truth Fuctions," American Mathematical Monthly, 62, 9, pp.627-631.

[4] R.A. Towle "Control and Data Dependence for Program Transformation," Ph. D. Dissertion, Report 76-788, Dept. of Computer Science, Univ. of Illinois at Urbana Champaign, Urbana, Illinios, 1976

[5] 황선명,진영택 "FORTRAN 벡터코드화를 위한 IF 변환 알고리즘," 대전대학교, 제8권 2호, pp.223-237,1990

[6] 이병관,이경환 "다중루프내에서의 종속심도 체크에 의한 벡타코드 생성." 정보과학회 논문지 18권 1호,1991

[7] Rider, Barbara Gershon "Incremental Data Flow Analysis based on A Unified Model of Elimination Algorithms," Ph.D dissertation, Rutgers University the State U. of New Jersey.

[8] F.K.Zadeck(1983) "Incremental Data Flow Analysis in a Structured Program Editor," Ph.D dissertation, Houston. Texas, 1983

황 선 명

1982년 중앙대학교 전자계산학과 졸업 (학사)
1984년 중앙대 대학원 전자계산학과 졸업 (석사)
1987년 중앙대 대학원 전자계산학과 졸업 (이학 박사)
1988년 독일 Bonn 대학 Informatik III post doctor
현재 대전대학교 전자계산학과 조교수
관심분야 : 소프트웨어 품질 보증 및 평가, 소프트웨어 테스팅

김 행 곤

1985년 중앙대학교 전자계산학과 졸업 (학사)
1987년 중앙대 대학원 이학 석사
1991년 중앙대 대학원 공학 박사
1978~1979년 미 항공우주국 연구원
1987~1990년 한국전기통신공사 전임연구원
1988~1989년 미 AT&T 연구원
1990~현재까지 효성여대 전자계산학과 조교수
관심분야 : 객체지향 시스템 설계, 사용자 인터페이스, 소프트웨어 재공학, 유지보수 자동화 툴, CASE