06장 제어 흐름

6.7 비결정성

알골 68에서 수식의 피연산자 간의 순서 정하기의 부재를 비결정성의 한 예로서 명시적으로 정의한다. 언어 설계자는 이를 평행한 실행이라 부른다. 다른 일부 고유 구성소는 알골 68에서 비결정적이다. 그리고 명시적인 평행한 문장은 프로그래머가 원할 때 임의의 수식의 계산에 있어서의 비결정성을 정하게 한다.

다익스트라[Dij75][1]는 선택과 논리 제어형에 대해 비결정성의 사용을 옹호했었다. 그의 보호 명령의 개념을 여러 언어에서 채택해왔다. 이 중 하나는 SR이다. 이를 12장에서 더 자세히 설명할 것이다. 잠시 두 정수 중 최대 값을 반환하는 함수를 작성하고 있다고 상상해보자. C에서 다음과 같은 간단한 코드를 사용할 수도 있다.

예 6.99

비결정성을 이용한 비대칭성 피하기

```
if (a > b) max = a;
else max = b;
```

물론 다음과 같이 작성할 수도 있다.

```
if (a >= b) max = a;
else max = b;
```

이 코드는 a=b일 때 그 행동이 다르다. 처음 코드는 max=b를 지정하지만 두 번째 코드는 max=a를 지정한다. 실질적으로 a와 b가 동일하기 때문에 이 차이는 중요하지 않다. 하지만 어떤 의미에서 둘 중 임의의 선택을 해야 하는 것은 심미적으로 불쾌하다. 더 중요한 것은 이런 선택의 임의성이 코드에 대해 추론하는 것이나 그것이 정확함

주1. 엣스거 다익스트라(1930~2002)는 병행성에 대한 현대적 이해의 논리적 기반 대부분을 개발했다. 또한 다른 많은 기여 중에 12.3.2절의 세마포어와 6.2절의 구조화된 프로그래밍 인자에 대해서도 많은 기여를 했다. 그는 1972년에 ACM 튜링 상(ACM Turing Award)을 받았다.

을 증명하는 것을 좀 더 어렵게 만든다. 보호 명령을 사용하는 언어(여기서의 예에서는 SR)에서 이렇게 작성할 수 있다.

예 6.100

보호 명령을 이용한 선택

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

이 구성소의 일반적인 형태는 다음과 같다.

```
if condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
fi
```

이 구성소 내의 각 조건을 **보호**라고 한다. 보호와 그것에 따라오는 문장을 함께 **보호 명령**이라 한다. 보호 명령을 사용하는 한 언어에서 제어가 `if`문에 도달하면 참으로 계산하는 보호 중 비결정적 선택을 하고 선택된 보호에 따라오는 문장 목록을 실행한다. SR에서 마지막 조건은 선택적으로 `else`가 될 수 있다. 조건 중 아무것도 참으로 계산될 수 없고 `else`가 있다면 그것 뒤에 따라오는 문장 목록을 실행한다. `else`가 없다면 `if`문 전체는 어떤 효력도 가지고 있지 않다(다익스트라의 본래 제안에서는 `else` 보호 선택권이 없다. 그리고 참이 되는 보호가 아무것도 없다면 그것은 동적 의미 오류다). 흥미롭게도 SR은 별개의 `case` 구성소를 제공하지 않는다. SR 컴파일러는 컴파일 시 상수의 중복되지 않는 집합에 대해 `if`문의 조건이 동일한 수식을 검사하는 시점을 찾아내어 적절하게 테이블 검색 코드를 만든다.

SR은 선택과 더불어 일부 목적을 위해 보호 명령을 사용한다. (또 다익스트라의 제안을 모방한) SR의 논리 제어형 루프 구성소는 `if`문과 흡사하다.

예 6.101

보호 명령을 사용한 루프 만들기

```
do condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
od
```

루프의 각 반복에 대해 참으로 계산되는 보호 중 비결정적인 선택을 하고 선택된 것에 따라오는 문장 목록을 실행한다. 보호 중 어느 것도 참이 되지 않으면(루프에 대한 `else` 보호 선택권이 없음) 루프는 종료된다. 이런 표기법을 사용하면 유클리드의 가장 유명한 공약수 알고리즘을 다음과 같이 작성할 수 있다.

```
do a > b -> a := a - b
[] b > a -> b := b - a
od
gcd := a
```

비결정적 병행성

예 6.102

비결정적인 메시지 수신

비결정적인 구성소가 미적과 형식 의미적 관점으로 볼 때에도 어느 정도 호소력을 갖지만 가장 주목할 만한 장점은 병행 프로그램에서 나타난다. 이런 프로그램에 대해 정확성에 영향을 줄 수 있다. 예를 들면 개인용 컴퓨터의 네트워크에 대해 컴퓨터를 사용한 설계를 지원하는 간단한 사전 프로그램을 작성하고 있다고 생각해보자. 그 사전은 부품 이름에서 명세서로의 매핑을 유지할 것이다. 사전 서버의 프로세스는 그 네트워크상에 있는 다른 워크스테이션의 클라이언트로부터의 요청을 다룬다. 각 요청은 읽기(부품 X에 대한 현재 명세서를 나에게 반환해라)나 쓰기(부품 Y를 아래와 같이 정의해라) 중에 하나일 것이다.[2] 클라이언트는 예측하지 못하는 시점에 요청을 보낸다. 결과적으로 서버는 어떤 주어진 시점에서 읽기나 쓰기 요청 중 어떤 것을 수신해야 하는지를 말할 수 없다. 잘못된 선택을 한다면 전체 시스템은 교착 상태에 빠질 수도 있다(©심화 학습에 있는 그림 6.8을 보라).

```
process client:
    loop
       동전을 던짐
       if 윗면이면 read 요청을 서버에게 보내고
          응답을 기다림
       if 뒷면이면 write 요청을 서버에게 보내고
          응답을 기다림

process server:
    loop
          read 요청을 받음
          자료와 함께 응답함
       OR
          write 요청을 받음
          자료와 응답을 갱신함
```

그림 6.8 | 비결정성을 필요로 하는 병행 프로그램의 예. 서버는 그 순간에 이용할 수 있는 것이 어떤 것이든지 read나 write 요청을 받아들일 수 있어야 한다. 특정 순서로 이를 수신해야 함을 주장하면 교착이 발생할지도 모른다.

예 6.103

SR에서 비결정적인 서버

대부분의 메시지 기반 병행 언어는 일부 가능한 통신 상대 중 누군가와의 통신을 정하기 위해 사용할 수 있는 적어도 한 개의 비결정적 구성소를 제공한다. SR에서 사전 서버를 다음과 같이 작성할 수 있다.

주2. 물론 이것은 과도하게 간소화된 예다. 무엇보다 이런 종류의 실제 시스템은 어느 것이나 그 사전상의 한 부분을 잠그기 위한 기법을 필요로 할 것이다. 그래서 어떤 두 클라이언트도 병행적으로 동일한 부분에 새로운 명세서를 설계하지는 못할 것이다.

```
# 요구된 유형의 선언:
op read_data(n : name) returns d : description
op write_data(n : name; d : description)

# 지역 서브루틴:
proc lookup ...          # 사전 내에서 정보를 찾음
proc update ...          # 사전 내의 정보를 변경함

# 서버를 위한 코드:
process server
    do true ->           # 무한 루프
       in read_data(n) returns d -> d := lookup(n)
       [] write_data(n, d) -> update(n, d)
       ni
    od
end
```

여기서 in은 통신 문장을 가질 수 있는 보호를 가지는 비결정적인 구성소다. 어떤 클라이언트가 한 부품에 대한 새로운 명세서를 포함한 요청을 전송해 시도하는 경우에 한해 보호 write_data(n, d)는 참으로 계산될 것이다. 12.4.3절에서 좀 더 정교화된 보호가 서버로 하여금 어떤 시점에서 수신하기를 원하는 요청의 유형을 제한하게 하거나 받아들일 수 있는지를 판단하기 위해 메시지 내부를 "엿보게" 함을 알 것이다. in문의 보호 중 어떤 것도 참이 아니라면 서버는 참이 될 때까지 기다린다.

보호의 선택

두 개 이상의 보호가 참으로 계산된다면 어떤 일이 발생할까? 그 언어 구현은 어떤 방법으로 하나를 선택할까? 지금까지 이 문제에 대해 그럴듯한 말을 해왔었다. 가장 나이브한 구현은 진부한 if...then...else와 같은 보호 명령 구성소를 다루는 것이다.

예 6.104

비결정성의 나이브(불공정한) 구현

```
server:
    loop
       if read_data 요청이 이용 가능하면
          ...
       elsif write_data 요청이 이용 가능하면
          ...
       else 어떤 요청이 이용 가능할 때까지 대기함
```

이 구현과 관련된 문제는 항상 다른 것보다 하나의 유형만을 선호한다는 것이다. read_data 요청이 항상 이용 가능하다면 write_data 요청을 결코 수신할 수 없을 것이다.

예 6.105

비결정성의 연속 구현에서 흔히 하는 실수

좀 더 세련된 구현은 보호 명령의 각 집합 안에 보호의 순환 리스트를 유지한다. 이 명령이 나타나는 구성소를 만날 때마다 마지막에 성공한 것 다음부터 시작해 보호를 검사할 것이다. 이 기법은 대다수의 경우 잘 동작하지만 어떤 경우에 시종 일관 실패할 수 있다. 예를 들면 (다시 SR로 작성된) 아래에서 첫 in문의 보호는 논리형 조건과 통신 검사를 결합시킨다.

```
process silly
var count : int := 0
    do true ->
       in A() st count % 2 = 1 -> ...
       [] B() -> ...
       [] C() -> ...
       ni
       count++
    od
```

이 예는 다소 인위적이지만 문제를 잘 설명한다. 첫 번째 보호에서 `st("such that")` 절은 루프의 홀수 번째 반복에서만 선택될 것임을 나타낸다. 이제 A, B와 C가 항상 이용 가능함을 가정하자. (처음에는 첫 번째 보호부터 시작해) 항상 마지막에 성공한 보호의 다음 것부터 검사한다면 첫 번째 반복에서는 (count mod 2 ≠ 1이기 때문에) B를 선택할 것이고, 두 번째 반복에서는 (count=2이면) C를, 세 번째 반복에서는 (다시 count mod 2 ≠ 1이기 때문에) B를 다시 선택한다. 이후 A를 결코 선택하지 않을 것이다. 이 예로부터 배워야 할 점은 어떤 결정적인 알고리즘도 비결정적 구성소를 통한 구현을 만족스럽지 않을 것이라는 점이다(밑의 보충설명을 보자).

마지막 쟁점은 부수효과와 관련된 것이다. 보호 명령 구성소는 참으로 계산되는 보호 중 비결정적 선택을 하게 한다. 하지만 선택을 하기 전에 보호 모두를 계산할 것임을 보장하지는 않는다. 일단 참인 보호를 선택하면 구현이 나머지 보호를 무시하는 것은 자유다.

그래서 보호 중 어떤 것이라도 부수효과를 가지면 프로그램은 기대하지 않거나 예상치 못한 결과를 만들어낼 수도 있다. 이 문제는 SR에서 프로그래머의 책임이다. 이에 대한 대안은 부수효과를 제한하고 컴파일러가 보호의 부재를 입증하게 되어 있다.

비결정성과 공정성

이상적으로 말하면 비결정적인 구성소에서 원하는 바는 공정성의 보장이다. 이는 기대하는 것보다 까다롭다는 것이 판명된다. "공정"을 정의하는 몇몇 그럴듯한 방법이 있다. 확실히 어떤 참인 보호도 항상 건너뛰지 않게 보장하길 원한다. 아마도 (반복의 가설적으로 무한한 나열에서) 매우 자주 참이 되는 보호 어느 것이나 항상 건너뛰지는 않기를 보장해주길 원한다. 한층 좋게 말하면 매우 자주 참이 되는 보호 모두를 매우 자주 선택하기를 요청할 것이다. 공정성의 더 강력한 개념은 참인 보호 사이의 선택이 진정으로 임의적이면 얻을 것이다. 운이 나쁘게도 좋은 의사 난수 생성기는 보호 간의 선택에 대해 그것을 선택하지 원하지 않을 수도 있을 만큼 충분히 비싸다. 결과적으로 보호 명령의 구현 대부분은 완전히 공정하지 않다. 대다수는 단지 순환 리스트 기법을 사용할 뿐이다. 다른 것은 다소 "좀더 임의적인" 휴리스틱을 사용한다. 예를 들면 대다수의 기계는 사용자 수준의 코드에서 효과적으로 읽을 수 있는 빠르게 변하는 클록 레지스터를 제공한다. 이 클록을 정수로 해석하고 보호의 개수로 이를 모듈라 연산해 나머지를 계산해 처음 계산될 보호를 합당하게 "임의로" 선택하게 할 수 있다.

✓ 확인문제

52. 보호 명령은 무엇인가?

53. 병행 프로그램에서 비결정성이 특히 중요한 이유는?

54. 비결정성에서 공정성의 다른 3가지 정의를 제시하라.

55. 참으로 계산되는 보호 중 하나의 선택을 구현하는 가능한 3가지 방법을 설명하라. 서로의 트레이드오프는 무엇인가?